FACULDADE DE ENGENHARIA DA UNIVERSIDADE DO PORTO

Comunicação Segura numa Plataforma de Simulação Distribuída

Davide Henrique Fernandes da Costa

DISSERTAÇÃO

Mestrado Integrado em Engenharia Informática e Computação

Supervisor: Daniel Castro Silva

Co-Supervisor: Daniel Garrido

Julho 23, 2020

Comunicação Segura numa Plataforma de Simulação Distribuída

Davide Henrique Fernandes da Costa

Mestrado Integrado em Engenharia Informática e Computação

Aprovado em provas públicas pelo Júri:

Presidente: Rui Carlos Camacho de Sousa Ferreira da Silva
Arguente: Paulo Jorge Pinto Leitão
Supervisor: Daniel Augusto Gama de Castro Silva

Julho 23, 2020

Abstract

The increasing number and diversity of autonomous vehicles has led the Artificial Intelligence and Computer Science Laboratory to develop a simulation platform where vehicles can be used to perform missions. Each platform entity like Air Traffic Controllers, Vehicles and Disturbances Manager is represented by an agent. To facilitate the communication between the different components of the platform, and using the concept of external agents, a multi-agent platform communication middleware was used, AgentService. However, this middleware uses polling to receive messages, making the communication mechanism very inefficient. Moreover, no security is used on the communication, making it possible for malicious agents to intercept and/or change the content of a message. The scale of simulations is increasing, and the communication is now a bottleneck for the evolution of the platform.

The main objective of this work is to introduce an efficient, fast and secure communication middleware in this simulation platform, while keeping AgentService existing features such as compliance with a standard agent communication language and support for integration of external applications. This middleware should improve: the performance (both latency and throughput), by changing the polling mechanism to an event-oriented one; and implement security on message delivery, guaranteeing confidentiality, integrity and authentication.

To achieve that, a study of the state of the art on multi-agent platforms, communication middleware and communication security was conducted. A comparative analysis, as well as some performance tests, were performed to decide which middleware or multi-agent platform to use. The tests showed that middleware platforms have better performance when compared with multi-agent platforms. RabbitMQ has proved to be the most consistent message oriented middleware during all tests and presented very good results with the use of security. Therefore, the developed solution uses RabbitMQ as the new middleware for the platform. This solution is capable of fulfilling all the requirements: previous AgentService features; and the enhancements, high performance and security on message delivery. This solution also manages agent distribution across several machines, making the system more suitable for computationally demanding scenarios.

To test our solution, we made performance testing on typical communication scenarios of the platform (but also of regular multi-agent systems) and we had very good results. The new middleware is capable of latencies hundreds of times lower, as well as dozens of times higher throughputs, when compared to AgentService middleware. The developed architecture should be generic enough so that it can be used for other multi-agent platforms.

Keywords: Multi-agent system, Communication middleware, Communication security, Distributed Simulation

Resumo

O aumento do número e da diversidade de veículos autónomos levou a que o Laboratório de Inteligência Artificial e Ciência de Computadores desenvolvesse uma plataforma de simulação de missões realizadas por um conjunto de veículos. Cada entidade da plataforma, como os Controladores de Tráfego Aéreo, os Veículos e o Gestor de Distúrbios, é representada por um agente. Para a comunicação entre os diferentes componentes e agentes da plataforma foi utilizado um *middleware* de comunicação de uma plataforma multi-agente, o AgentService. No entanto, este *middleware* utiliza *polling* na receção de mensagens, o que torna o mecanismo bastante ineficiente. Para além disso, as mensagens são trocadas sem qualquer tipo de segurança, o que possibilita que agentes maliciosos intercetem e/ou alterem o conteúdo de uma mensagem. O tamanho dos cenários de simulação está a aumentar, sendo neste momento a comunicação o *bottleneck* para a evolução da plataforma.

O objetivo principal desta dissertação é a introdução de um *middleware* de comunicação eficiente, rápido e seguro na plataforma de simulação, mantendo o suporte, vindo do AgentService, a uma linguagem padrão de comunicação entre agentes e à integração de aplicações externas. Este *middleware* deve conseguir melhorar o desempenho (latência e taxa de mensagens enviadas), alterando o mecanismo de comunicação para um orientado a eventos; e implementar segurança na troca de mensagens, garantindo confidencialidade, integridade e autenticidade.

Para isso, foi realizado um estudo do estado da arte em plataformas multi-agente, *middleware* de comunicação e segurança na comunicação. Uma análise comparativa, bem como alguns testes de desempenho, foram realizados para decidir qual o *middleware* de comunicação ou plataforma multi-agente a usar. Os testes mostraram que os *middleware* apresentam melhor desempenho quando comparados com as plataformas multi-agente. O RabbitMQ foi o *middleware* orientado a mensagens mais consistente ao longo dos testes e apresentou resultados bastante bons com o uso de segurança. Assim, a solução desenvolvida usa o RabbitMQ como o novo *middleware* da plataforma. A solução é capaz de cumprir todos os requisitos: aqueles já existentes no AgentService; e as melhorias, elevado desempenho e segurança na troca de mensagens. Esta solução permite também a distribuição dos agentes por várias máquinas, possibilitando o uso do sistema para cenários de simulação computacionalmente mais exigentes.

De forma a testar a solução desenvolvida, realizamos testes de desempenho em cenários típicos de comunicação da plataforma do LIACC (mas também de um qualquer sistema multi-agente) e obtivemos resultados bastante bons. O novo *middleware* é capaz de latências centenas de vezes mais baixas, bem como taxas de mensagens enviadas dezenas de vezes maiores, quando comparado com o AgentService. A arquitetura desenvolvida deve ser genérica o suficiente para permitir a sua integração noutras plataformas multi-agente.

Palavras-Chave: Sistema Multi-Agente, *Middleware* de Comunicação, Segurança na Comunicação, Simulação Distribuída

Agradecimentos

Gostaria de começar por agradecer ao meu orientador, Daniel Silva, e co-orientador, Daniel Garrido, pela ajuda que sempre me disponibilizaram e pelo conhecimento que me transmitiram.

Não poderia deixar de agradecer à minha família e amigos mais próximos pelo apoio neste novo e grande projeto, estando sempre lá quando precisei, como aliás em toda a minha vida.

Davide Costa

"The journey of a thousand miles must
begin with a single step."

Lao Tzu

Conteúdo

Lista de Figuras

Lista de Tabelas

Acrónimos

ACC	*Agent Communication Channel* (Canal de Comunicação de Agentes)
ACL	*Agent Communication Language* (Linguagem de Comunicação de Agentes)
AER	*AgentService External Runtime*
AMQP	*Advanced Message Queuing Protocol* (Protocolo Avançado de Enfileiramento de Mensagens)
API	*Application Programming Interface* (Interface de Programação de Aplicações)
AMS	*Agent Management System* (Sistema de Gestão de Agentes)
CLI	*Common Language Infrastructure* (Infraestrutura de Linguagem Comum)
DAI	*Distributed Artificial Intelligence* (Inteligência Artificial Distribuída)
DDS	*Data Distribution Service* (Serviço de Distribuição de Dados)
DF	*Directory Facilitator* (Facilitador de Diretórios)
DM	*Disturbances Manager* (Gestor de Distúrbios)
FEUP	*Faculdade de Engenharia da Universidade do Porto*
FIPA	*Foundation for Intelligent Physical Agents* (Fundação para os Agentes Físicos Inteligentes)
FSX	*Flight Simulator X*
GUI	*Graphical User Interface* (Interface Gráfica do Utilizador)
HTTP	*Hyper Text Transfer Protocol* (Protocolo de Transferência de Hipertexto)
HTTPS	*Hyper Text Transfer Protocol Secure* (HTTP Seguro)
IMTP	*Internal Message Transport Protocol* (Protocolo Interno de Transporte de Mensagens)
IP	*Internet Protocol* (Protocolo da Internet)
IoT	*Internet of Things* (Internet das Coisas)
IRC	*Internet Relay Chat* (Comunicação Retransnmitida pela Internet)
JIAC	*Java-based Intelligent Agent Componentware* (*Componentware* de Agentes Inteligentes baseada em Java)
JADE	*Java Agent DEvelopment Framework* (Ferramenta de Desenvolvimento de Agentes em Java)
JMS	*Java Message Service* (Serviço de Mensagens do Java)
JSON	*JavaScript Object Notation* (Notação de Objeto JavaScript)
KQML	*Knowledge Query and Manipulation Language* (Linguagem de Manipulação e Consulta de Conhecimento)
LIACC	Laboratório de Inteligência Artificial e Ciência de Computadores
MAS	*Multi-Agent System* (Sistema Multi-Agente)
MIME	*Multipurpose Internet Mail Extensions* (Extensões Multi função para Mensagens de Internet)
MOM	*Message Oriented Middleware* (*Middleware* Orientado a Mensagens)

MQTT	*Message Queuing Telemetry Transport* (Transporte de Telemetria de Enfileiramento de Mensagens)
MSMQ	*Microsoft Message Queue* (Fila de Mensagens Microsoft)
MUC	*Multi-user Conference* (Conferência Multi-Utilizador)
OMG	*Object Management Group*
OpenMAMA	*Open Middleware Agnostic Messaging API* (API de Mensagens Independentes de *Middleware*)
ORB	*Object Request Broker* (Intermediário de Solicitação de Objetos)
P2P	*Peer-to-Peer* (Nó-para-Nó)
QoS	*Quality of Service* (Qualidade de Serviço)
REST	*Representational State Transfer* (Transferência Representacional de Estado)
RestMS	*RESTful Messaging Service* (Serviço de Mensagens seguindo o Modelo REST)
RMI	*Remote Method Invocation* (Invocação Remota de um Método)
RPC	*Remote Procedure Calls* (Chamada Remota de um Procedimento)
RSA	*Rivest-Shamir-Adleman*
SASL	*Simple Authentication and Security Layer* (Camada Simples de Autenticação e Segurança)
SDK	*Software Development Kit* (Kit de Desenvolvimento de Software)
SPADE	*Smart Python Agent Development Environment* (Ambiente de desenvolvimento de Agentes Inteligentes em Python)
SOA	*Service-Oriented Architecture* (Arquitetura Orientada a Serviços)
SOAP	*Simple Object Access Protocol* (Protocolo de Acesso de Objeto Simples)
SSL	*Secure Socket Layer*
STOMP	*Simple Text Oriented Messaging Protocol* (Protocolo de Mensagens Orientado a Texto Simples)
TCP	*Transmission Control Protocol* (Protocolo de Controle de Transmissão)
TLS	*Transport Layer Security*
UAV	*Unmanned Aerial Vehicle* (Veículo Aéreo Não Tripulado)
UDP	*User Datagram Protocol* (Protocolo de Datagrama do Utilizador)
UMAP	*Universal Multi-Agent Platform for .NET developers* (Plataforma Multi-Agente Universal para programadores .NET)
UxAS	*Unmanned Systems Autonomy Services* (Serviços de Autonomia para Sistemas Não Tripulados)
WCF	*Windows Communication Foundation* (Fundação de Comunicação Windows)
XML	*Extensible Markup Language* (Linguagem de Notação Extensível)
XMPP	*Extensible Messaging and Presence Protocol* (Protocolo de Mensagens Extensíveis e de Presença)

Capítulo 1

Introdução

Esta dissertação do Mestrado em Engenharia Informática e Computação foca-se na comunicação segura e eficiente entre agentes num sistema multi-agente. O uso crescente de sistemas multi-agente, nos mais diversos cenários e aplicações, levou à necessidade de plataformas multi-agente que usem *middleware* de comunicação mais seguros e eficientes [Leitão et al., 2013, Pires et al., 2018]. Estes *middleware* devem permitir comunicação próxima de tempo real, com baixas latências e elevadas taxa de mensagens enviadas, permitindo assegurar confidencialidade, integridade e autenticidade na troca de mensagens [Briones et al., 2016, Calvaresi et al., 2017].

Contexto e Motivação

Nos últimos anos têm vindo a aumentar o número e a diversidade de veículos autónomos usados em diferentes tipos de tarefas. Assim, com vista à realização de missões conjuntas por um conjunto de veículos heterogéneos (aviões, carros, barcos e submarinos), o Laboratório de Inteligência Artificial e Ciência de Computadores (LIACC) tem vindo a desenvolver uma plataforma que permite a simulação deste tipo de missões [Silva, 2011].

De forma a facilitar a comunicação entre os diferentes componentes da plataforma, foi usado um sistema multi-agente, o AgentService, que respeita as normas definidas pela FIPA[1] (*Foundation for Intelligent Physical Agents*) para o efeito. Este Sistema Multi-Agente (MAS, do inglês *Multi-Agent System*) permite a integração de agentes externos, isto é, permite que uma aplicação desenvolvida de forma independente e externa à plataforma MAS possa registar-se como agente na plataforma, acedendo às funcionalidades típicas dos agentes [The AgentService Team, 2008]. Estas funcionalidades incluem, por exemplo, o envio e receção de mensagens e o acesso às páginas brancas e amarelas. A plataforma do LIACC usa estas funcionalidades para a comunicação e descoberta de cada uma das suas aplicações/agentes.

No entanto, o *middleware* responsável pela comunicação entre agentes externos usa um método de *polling* para receção de mensagens, degradando substancialmente o seu desempenho, e

[1] Mais informação disponível em `http://fipa.org/`

fazendo com que o *middleware* do AgentService se tenha tornado no principal *bottleneck* da plataforma do LIACC [Almeida, 2017].

Para além disso, a comunicação não utiliza qualquer tipo de segurança, deixando a plataforma vulnerável a agentes maliciosos. Um agente malicioso pode, por exemplo, fazer-se passar por um outro agente, enviando mensagens em seu nome. Pode também interpretar as mensagens trocadas entre outros agentes, ou até alterar o conteúdo de uma mensagem.

Objetivos e Questões de Investigação

O principal objetivo desta dissertação é a introdução de melhoramentos, ao nível da comunicação, na plataforma de simulação existente. Estas melhorias na comunicação focam-se em:

- **Desempenho:** permitindo uma menor latência entre o envio e a receção de uma mensagem, bem como uma maior taxa de transferência de mensagens.

- **Segurança:** permitindo assegurar integridade, confidencialidade e autenticidade na troca de mensagens.

Estas duas melhorias são, de certa forma, contrárias devido ao impacto negativo no desempenho da plataforma que o uso de segurança na troca de mensagens poderá trazer. No entanto, como sabemos que o AgentService é um *middleware* bastante ineficiente, espera-se que mesmo assim o desempenho seja melhorado. Caso isto não aconteça, ou para situações em que o desempenho seja uma prioridade, será ainda estudado o *tradeoff* entre segurança e desempenho. A plataforma poderá assim apresentar várias versões com níveis de segurança diferentes. A ideia é que a plataforma possa utilizar segurança na maioria das situações (com todas as garantias ou apenas parte delas), mas que em situações em que as necessidades de desempenho são máximas e a segurança não é um fator importante, se possam desligar totalmente os mecanismos de segurança.

Espera-se que a solução produzida, bem como uma descrição detalhada do processo que lhe deu origem, permita a introdução destas melhorias noutras plataformas semelhantes.

Para atingir os objetivos propostos, serão investigadas as seguintes questões, que esperamos conseguir responder com o desenvolvimento desta dissertação:

- **RQ1:** Será possível encontrar um *middleware* que permite a comunicação segura mantendo um elevado desempenho?

- **RQ2:** Será que a integração de um *middleware* num MAS consegue assegurar comunicação mais eficiente e segura?

- **RQ3:** Será possível o desenvolvimento de uma arquitetura genérica que permite a integração deste *middleware* na plataforma do LIACC, mas também numa qualquer plataforma multi-agente?

Metodologia e Resultados esperados

No sentido de tentar solucionar o problema proposto, será feita uma análise da literatura de forma a encontrar informação sobre o que de melhor foi feito na área, que soluções existem e quais as vantagens e desvantagens que apresentam. Esta análise foca-se essencialmente em três pontos: plataformas multi-agente, *middleware* de comunicação e mecanismos de segurança.

O estudo das plataformas multi-agente existentes no mercado permitirá perceber se existe alguma que vá de encontro ao que se procura, ou caso não existam, o que está em falta.

O estudo dos *middleware* de comunicação surge numa fase seguinte, de forma a perceber aquilo que existe no âmbito da comunicação em particular, e que não está presente nas plataformas multi-agente analisadas, sendo benéfica a sua integração no sistema multi-agente utilizado na solução produzida.

O estudo dos mecanismos de segurança e das suas caraterísticas irá permitir perceber que garantias trazem e como podem ser utilizados para obter as garantias pretendidas, isto é, integridade, confidencialidade e autenticidade na troca de mensagens.

De forma a ajudar na decisão de que plataforma ou que conjugação de plataforma e *middleware* utilizar, será necessário perceber o desempenho das diferentes soluções. Para isso será necessário um conjunto de testes no contexto de uso da plataforma. As soluções serão avaliadas com e sem o uso de segurança para conseguir perceber o impacto do uso de segurança em cada uma delas, mas também de forma geral.

A solução encontrada, quer seja a integração de um novo *middleware* de comunicação no AgentService, quer seja a integração de uma nova plataforma multi-agente, requer a adaptação e posterior integração na plataforma do LIACC. Espera-se obter uma solução adaptada à comunicação típica de uma plataforma multi-agente, com um *middleware* de comunicação rápido e seguro. Para além disso, este *middleware* deve manter a interoperabilidade existente na plataforma do LIACC, e importante para qualquer sistema multi-agente, nomeadamente: o suporte FIPA e o suporte para integração de agentes externos, bem como apresentar as melhorias introduzidas. Estas melhorias passam por tornar a comunicação da plataforma mais eficiente, rápida e segura. Esta solução deverá poder ser integrada na plataforma do LIACC, mas também noutras plataformas do mesmo género.

A solução produzida irá ser avaliada recorrendo, novamente, a testes de desempenho usando casos de uso típicos da plataforma de simulação do LIACC, comparando o seu desempenho antes e depois da implementação da solução. Estes casos de uso incluem comunicação um-para-um entre 2 agentes, comunicação em *broadcast* e comunicação muitos-para-um. Os testes de desempenho realizados devem mostrar uma melhoria global do desempenho de toda a plataforma, que estava limitada pela baixa capacidade e caraterísticas do *middleware* de comunicação usado.

Em termos de segurança, a comunicação deverá apresentar confidencialidade, integridade e autenticidade, permitindo que a plataforma esteja segura contra ataques que seriam, até à integração da solução, facilmente realizados.

Estrutura do Documento

O restante documento é organizado da seguinte forma: o capítulo 2 apresenta uma contextualização sobre a área desta dissertação, de forma a acompanhar o resto do documento. Este capitulo contém um estudo da funcionalidade e arquitetura da plataforma de simulação do LIACC, das plataformas para sistemas multi-agente e a sua comunicação, dos diferentes tipos de *middleware* de comunicação (com especial foco em *middleware* orientados a mensagens) e segurança na comunicação e em sistemas multi-agente; No capítulo 3 apresenta-se uma revisão da literatura em vários temas relacionados com o trabalho a desenvolver, focado em plataformas para sistemas multi-agente e num conjunto de plataformas *message oriented middleware*; No capítulo 4 apresenta-se uma arquitetura base para a solução desenvolvida, tendo em conta os testes de desempenho realizados também nesse capítulo. Para além disso é feita uma descrição mais detalhada do problema e das opções de solução possíveis; No capítulo 5 apresenta-se todo o trabalho desenvolvido, nomeadamente a substituição do AgentService pelo RabbitMQ, a implementação de comunicação segura com o RabbitMQ e a distribuição dos agentes na plataforma de simulação; No capítulo 6 apresentam-se um conjunto de testes que têm como objetivo a avaliar a solução desenvolvida; No capítulo 7 apresenta-se um sumário de tudo o que foi dito e realizado, algumas conclusões, limitações e trabalho futuro.

Capítulo 2

Contextualização

Este capítulo de contextualização fornece um conjunto de conceitos e fundamentos que servem de base ao restante documento. Está dividido em cinco grupos principais - plataforma de simulação do LIACC, sistemas multi-agente, *middleware* de comunicação, *middleware* de comunicação orientados a mensagens e segurança na comunicação.

2.1 Plataforma de Simulação do LIACC

A plataforma do LIACC permite a simulação de missões realizadas por um conjunto heterogéneo de veículos (aviões, automóveis, barcos e submarinos). Estas missões podem ser dos mais variados tipos, tirando partido do uso de todos os veículos, separados ou em conjunto. Assim, alguns exemplos de utilização da plataforma são: localização e salvamento de pessoas (por exemplo perdidas ou em risco numa montanha com difícil visibilidade e acesso); deteção de incêndios através do patrulhamento de florestas; deteção de aberturas hidrotermais na fundo do mar; deteção de fontes de poluição e avaliação dos níveis de poluição emitidos; deteção de um alvo em movimento e consequente perseguição, por exemplo para fins militares; deteção de atividades ilegais como limpeza de tanques de petróleo em alto mar ou plantações de drogas ilegais [Silva, 2011].

A arquitetura da plataforma, Fig. 2.1, é composta por diversos componentes que funcionam em conjunto para permitir todas as funcionalidades descritas em [Silva, 2011] e [Almeida, 2017]. Os componentes da plataforma são descritos a seguir.

- **Control Panel (Painel de Controlo)** é a entidade central do sistema, responsável por toda a configuração do sistema, ambiente, perturbações existentes, lançamento de agentes em execução, criação e processamento de missões e ainda por fornecer informação sobre o estado global do sistema durante a realização da missão.

- **ATC Agent (Agente de Controlo de Tráfego Aéreo)** funciona como um típico controlador de tráfego aéreo que vemos nos aeroportos, mas neste caso com responsabilidade não só de monitorização do tráfego aéreo, mas também terrestre e marítimo. Podem existir vários

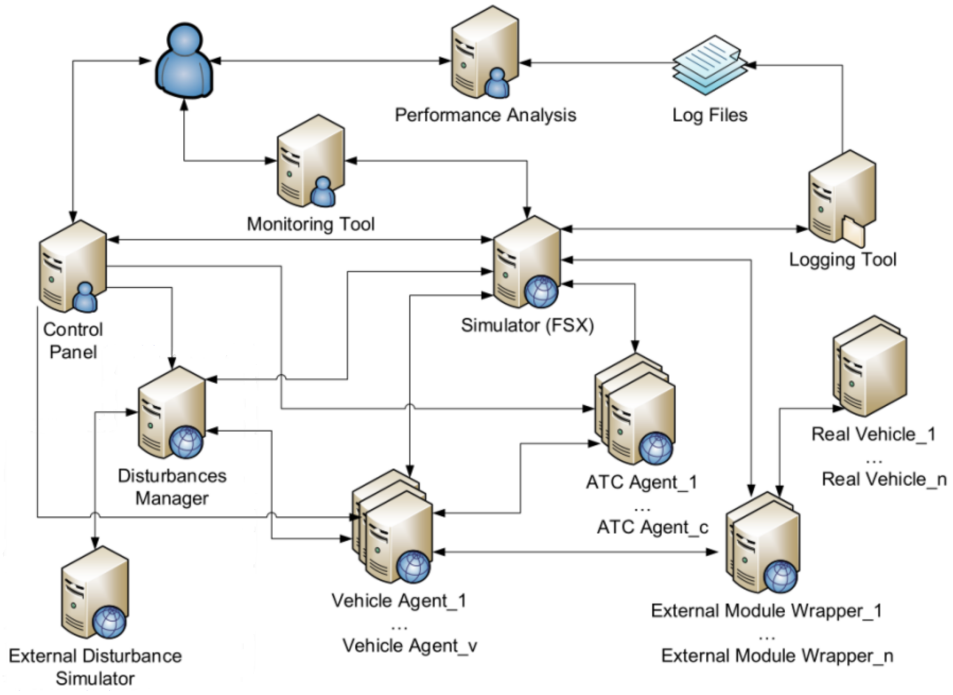


Figura 2.1: Arquitetura Geral da Plataforma de Simulação do LIACC (Adaptado de [Almeida, 2017])

destes agentes, sendo que cada um será responsável por uma determinada área e/ou tipo de tráfego (aéreo, terrestre ou marítimo).

- **Vehicle Agent (Agente Veículo)** representa um agente que funciona como um veículo na plataforma. Este agente, criado pelo *Control Panel*, é responsável por controlar, por exemplo, a sua navegação e evitar acidentes.

- **Real Vehicle (Veículo Real)** representa um veículo real que pode ser integrado na plataforma como um *Vehicle Agent* através do *External Module Wrapper*. Os dados deste veículo real serão utilizados na simulação, juntamente com outros veículos, reais ou virtuais.

- **Disturbances Manager (Gestor de Perturbações)** permite a gestão dos distúrbios (por exemplo um incêndio) que não possam ser simulados pela plataforma de simulação escolhida (FSX). É ainda responsável pela comunicação com os veículos, para que eles tenham informação nos seus sensores sobre esses distúrbios.

- **External Disturbances Simulator (Simulador de Perturbações Externas)** permite a ligação de um simulador externo ao *Disturbances Manager*. Este simulador poderá fornecer uma simulação mais exata para um determinado desastre.

- **Monitoring Tool (Ferramenta de Monitorização)** permite a visualização do estado da simulação em tempo real.

- **Logging Tool (Ferramenta de Registos)** permite registar em ficheiros os dados relativos a cada missão, com estado inicial, comunicação entre os agentes e descrição do estado de cada agente. Estes ficheiros produzidos (*Log Files*) permitem a análise de desempenho, usando o módulo *Performance Analysis* (Análise de Desempenho).
- **FSX (Flight Simulator X)** é o simulador utilizado pela plataforma. Este simulador, desenvolvido pela Microsoft, é o responsável pela simulação dos veículos e da sua movimentação.

Na plataforma, uma simulação típica começa pela definição do cenário (como as bases de operações e zonas de voo restrito) e a criação dos agentes ATC. De seguida é feita a especificação da equipa e criação dos agentes veículo, bem como a especificação dos distúrbios e criação do *Disturbances Manager*. Finalmente é feita a especificação da missão, e envio dessa informação para os agentes veículo, que terão de comunicar entre si, com os ATCs e com o *Disturbances Manager* durante a missão. Assim, para a comunicação entre as várias entidades, surge a necessidade de utilizar um *middleware* de comunicação. Neste caso, foi utilizada uma plataforma multi-agente, o AgentService, e o seu *middleware* de comunicação integrado. A escolha do AgentService baseou-se na possibilidade de este apresentar suporte para C# com agentes externos, capacidade de executar em federação e ser uma plataforma MAS que segue as normas FIPA [Silva, 2011]. Na plataforma do LIACC, o AgentService tem como principal funcionalidade fornecer capacidade de comunicação entre os diversos agentes/aplicações da plataforma segundo as normas FIPA e capacidade de descoberta de outros agentes através dos serviços de páginas amarelas e brancas.

Recentes desenvolvimentos no *Disturbances Manager* (DM) vieram trazer uma grande necessidade de comunicação com os agentes veículo [Almeida, 2017]. O DM é responsável pelo envio, aos agentes veículo da plataforma, das informações dos seus sensores sobre os distúrbios existentes (por exemplo as leituras de temperatura quando na presença de um incêndio). Neste sentido, na presença de vários distúrbios e vários agentes, é necessário um *middleware* de comunicação que permita o envio da taxa de mensagens necessária e que consiga manter essa taxa ao longo do tempo. Para além disso, os sensores podem ter várias taxas de atualização, pelo que devemos garantir capacidade para as taxas mais elevadas. O uso do AgentService para esta comunicação é um *bootleneck* para o desempenho do DM [Almeida, 2017].

2.2 Sistemas Multi-Agente

Nos últimos anos, houve um aumento do interesse por Inteligência Artificial Distribuída (DAI, do inglês *Distributed Artificial Intelligence*) devido à sua capacidade de resolução de problemas complexos. A DAI pode ser dividida em três grandes áreas [Dorri et al., 2018]:

Inteligência Artificial Paralela que consiste no desenvolvimento de algoritmos, linguagens e arquiteturas paralelas que permitam melhorar a eficiência dos algoritmos utilizados pela inteligência artificial clássica, através da paralelização das tarefas de processamento.

Resolução de Problemas de forma Distribuída que consiste na resolução de problemas através da divisão destes em problemas de menor dimensão. Cada sub-problema é depois resolvido

por uma entidade computacional autónoma, um agente. No entanto, estes agentes envolvidos apenas cooperam no conhecimento, recursos e alguma comunicação pré-definida, o que limita a flexibilidade dos agentes e do sistema.

Sistemas Multi-Agente que consistem num conjunto de entidades computacionais autónomas, a que se chamam agentes. Os agentes são capazes de agir por conta própria ou através da interação com outros. Ao interagirem com outros agentes e com o ambiente em que estão inseridos, os agentes vão recolher novos contextos e ações. Com esta informação, e usando agora o seu raciocínio, os agentes tomam decisões e realizam ações no ambiente rumo ao seu objetivo. Os agentes podem ter objetivos comuns, nesse caso estamos perante uma coordenação cooperativa. No entanto, os agentes podem ter objetivos diferentes e, nesse caso, estamos perante uma coordenação competitiva (por exemplo num sistema de leilões, em que cada agente é um licitador). Os sistemas multi-agente têm evoluído devido à necessidade de resolver problemas cada vez mais complexos e que seguem uma estrutura necessariamente distribuída, seja pelo problema ser demasiado complexo para uma solução centralizada, seja pela necessidade de manter certos conhecimentos restritos ao agente ou pela possibilidade de escalabilidade [Anumba et al., 2005].

Num sistema multi-agente, a comunicação é então um mecanismo fundamental. A comunicação permite aos agentes interagirem uns com os outros, no sentido de atingirem os seus objetivos. Sem a comunicação estes estariam totalmente isolados do mundo, fechados no seu ciclo de perceção-deliberação-ação [Michel et al., 2009].
Com o estudo de mecanismos de comunicação entre agentes durante mais de 50 anos, três principais surgiram [Dorri et al., 2018]:

- **Speech act**, em que um agente emite uma afirmação que altera o ambiente em que está inserido. Por exemplo a afirmação "Declaro-vos marido e mulher", produz efeitos nos papéis de 2 outros agentes envolvidos [Kibble, 2006].

- **Message Passing**, em que um agente envia uma mensagem para um outro agente ou para um conjunto de agentes.

- **Blackboard**, em que é utilizado um repositório central de informação, onde cada agente coloca a informação que pretende partilhar e recolhe informação partilhada pelos outros agentes.

O modelo utilizado na maioria dos sistemas multi-agente, para a comunicação entre agentes, é Message Passing (troca de mensagens) [Michel et al., 2009]. Estas mensagens precisam, no entanto, de apresentar uma semântica comum, que seja reconhecida por todos os intervenientes da comunicação. Por exemplo para uma mensagem que pretende indicar a temperatura como 20 (graus Celsius), é necessário que os agentes recetores interpretem a temperatura em graus Celsius(^{o}C) e não Fahrenheit (^{o}F) ou Kelvin (K). Assim, surge a necessidade de existência de uma linguagem padrão que todos os agentes utilizem para construção e interpretação das mensagens trocadas, a *Agent Communication Language* (ACL). Ao longo dos anos várias implementações

surgiram, sendo neste momento a linguagem FIPA ACL[1] a mais utilizada [Dorri et al., 2018, Michel et al., 2009]. Esta linguagem padrão de comunicação entre agentes, similar e derivada da KQML (*Knowledge Query and Manipulation Language*) [Chalupsky et al., 1992], contém um conjunto de parâmetros como emissor, recetor, conteúdo e performativa.

A FIPA (*Foundation for Intelligent Physical Agents*) é uma organização internacional que promove o desenvolvimento de técnicas padrão no desenvolvimento de sistemas multi-agente. A sua especificação refere um modelo de referência para o desenvolvimento de uma plataforma multi-agente. Este modelo, que pode ser visto na Fig. 2.2, é constituído por [FIPA TC Agent Management, 2004]:

- O **AMS (*Agent Management System*)** que funciona como supervisor da plataforma, controlando o acesso e uso da plataforma por parte dos agentes. Cada agente deve registar-se perante o AMS, permitindo que este funcione como um serviço de páginas brancas com os nomes dos agentes registados, a que outros agentes podem aceder.
- O **MTS (*Message Transport Service*)** que é responsável pelo transporte de mensagens FIPA ACL entre agentes na mesma plataforma ou em plataformas diferentes.
- O **DF (*Directory Facilitator*)** que funciona como um serviço de páginas amarelas para outros agentes. Os agentes registam-se na plataforma indicando os serviços que fornecem, possibilitando aos outros agentes encontrar os serviços fornecidos. Este serviço é opcional.

Uma plataforma multi-agente diz-se compatível com as normas FIPA quando segue o modelo de referência FIPA apresentado [Obitko, 2007].

Como mostrado na Fig. 2.2, a comunicação entre os agentes de uma plataforma multi-agente acontece através do serviço de transporte de mensagens (MTS). Este serviço permite o transporte entre diferentes plataformas e assim o funcionamento em federação. Uma federação de plataformas multi-agente consiste num conjunto de instâncias de uma dada plataforma a trabalharem em conjunto, permitindo a distribuição dos agentes pela várias máquinas em que as plataformas estão em execução [Wang and Zhang, 2012].

As plataformas multi-agente têm como objetivo ajudar no desenvolvimento de sistemas multi-agente permitindo abstrair algumas funcionalidades como a comunicação e a distribuição. Várias plataformas foram desenvolvidas pela comunidade ao longo dos anos, seguindo total ou parcialmente as normas FIPA, fornecendo os serviços básicos e importantes para todos os sistema multi-agente [Sujil et al., 2018]. Na secção 3.1 apresentamos uma análise detalhada a algumas das plataformas desenvolvidas.

2.3 *Middleware* de comunicação

Várias definições de *middleware* podem ser encontradas na literatura [Mohamed et al., 2008, Elkady and Sobh, 2012], sendo que uma das mais aceites é: Um *middleware* é um software de

[1] Mais informação disponível em `http://fipa.org/specs/fipa00061/index.html`

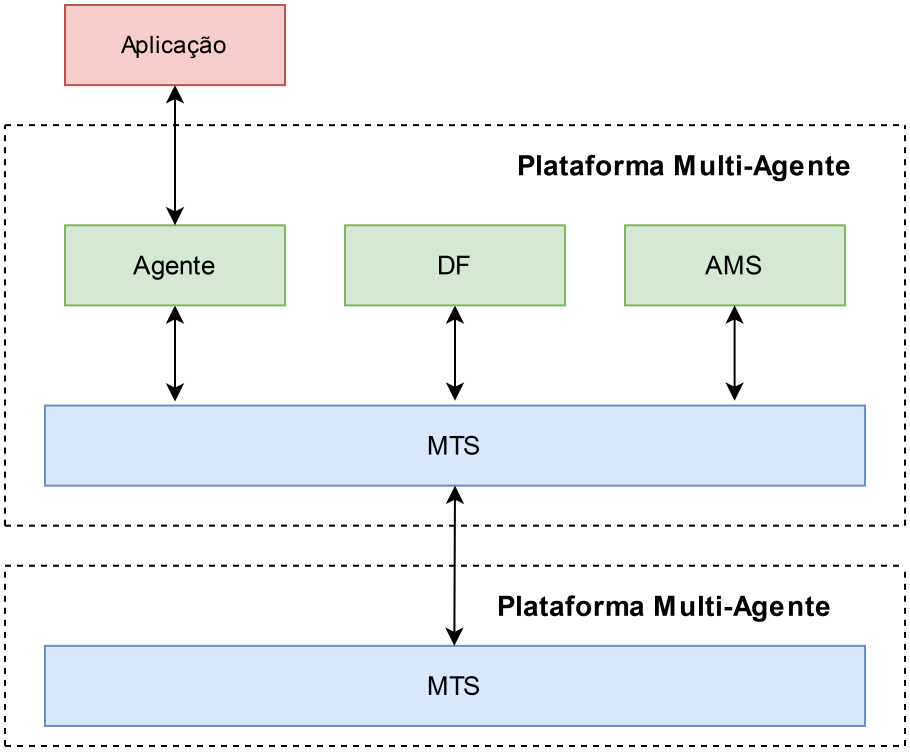


Figura 2.2: Modelo de Referência FIPA (Adaptado de [Obitko, 2007])

comunicação que permite a vários processos, a correr numa ou várias máquinas, interagirem uns com os outros [Fischer et al., 2016]. Existem três tipos de *middleware*:

- ***Message Oriented Middleware (MOM)*** é uma família de software que permite a partilha de mensagens entre nós na rede. O objetivo deste tipo de *middleware* é abstrair os pormenores técnicos da ligação e dos serviços que interligam toda a rede [Fischer et al., 2016]. O envio e receção de mensagens, entre diferentes nós da rede, é feito de forma completamente assíncrona. Este *middleware* usa filas para onde as mensagens são enviadas. Um nó envia uma mensagem, especificando o identificador único da fila na qual a mensagem deve ser colocada. Um nó recetor lê depois a mensagem da fila.

- ***Remote Procedure Calls (RPC)*** permite invocar métodos num computador remoto. Este, que é o mais antigo tipo de *middleware* [Saghian and Ravanmehr, 2014], pode ser usado no modelo cliente-servidor com diferentes aplicações, como por exemplo para um cálculo intensivo que é requerido por um cliente. O cliente faz uma chamada remota ao servidor, que realiza o cálculo por ter maior poder de computação, e retorna o resultado ao cliente. O retorno do resultado ao cliente era, nas versões iniciais de RPC, obrigatoriamente síncrono, mas surgiram depois versões assíncronas [Ananda et al., 1995, Fischer et al., 2016].

- ***Object Request Broker (ORB)*** permite a comunicação entre objetos. Assim, promove interoperabilidade permitindo aos utilizadores criar objetos que comunicam entre si usando

o ORB. O ORB fornece uma estrutura que permite a localização de objetos na rede e operações de chamadas nestes objetos, como se os objetos remotos estivessem localizados no mesmo processo do cliente, fornecendo transparência de local. Este *middleware* tem como base o RPC mas apresenta carateristicas orientadas a objetos, como *inheritance* e possibilidade de uso de referências remotas e exceções [Saghian and Ravanmehr, 2014].

No âmbito desta dissertação, e tendo em conta que a forma tradicional de comunicação entre agentes utiliza mensagens [Michel et al., 2009], foi aprofundado o estudo dos *Message Oriented Middleware (MOM)*.

2.4 *Middleware* Orientados a Mensagens

O uso deste *middleware* tem crescido bastante nos últimos anos com a necessidade crescente de tecnologias adequadas à distribuição. Esta tecnologia foca-se num mecanismo de troca de mensagens, também chamado de padrão de fila de mensagens, usando dois conceitos principais: o conceito de canal por onde circulam mensagens e o conceito de fila onde são guardadas as mensagens entre a escrita e a leitura [Celar et al., 2016]. O uso de filas de mensagens permite uma interação completamente assíncrona.

As filas de mensagens (também chamadas de filas de espera) têm várias formas de conservação de mensagens:

- Por consumo, em que a mensagem é mantida até ser lida por um consumidor. Depois de lida, esta é eliminada da fila de espera.
- Por tempo, em que a mensagem é mantida durante um tempo predefinido. Isto é, mesmo que lida a mensagem é mantida e apenas eliminada ao fim do tempo especificado. No caso de não ter sido lida a mensagem será, de qualquer forma, eliminada quando o tempo expirar.
- Por tempo e consumo, em que a mensagem é mantida durante o tempo predefinido ou até ser lida por um consumidor, isto é, até que a primeira das condições aconteça. Este sistema funciona como um híbrido dos 2 anteriores.

O consumo destas mensagens pode ser feito baseado em dois princípios [Marcos, 2016]:

- ***FIFO - First In First Out***, em que as mensagens são consumidas na mesma ordem pela qual foram produzidas (isto é, colocadas na fila à espera de serem processadas).

- ***FIFO Priorities - First In First Out with Priorities***, em que como em FIFO as mensagens são consumidas na mesma ordem pela qual foram produzidas, existindo, no entanto, um novo conceito de prioridade associado à mensagem. Uma mensagem pode ser elevada na fila de prioridade, isto é, ser colocada mais próxima da saída da fila, se o seu grau de prioridade, normalmente um número inteiro num campo de meta-dados associado à mensagem, for superior ao de outras mensagens à espera de serem consumidas.

Um consumidor fica constantemente à escuta de mensagens numa fila, que é preenchida com mensagens de um produtor. Uma fila pode ter vários produtores a preenchê-la, e vários consumidores podem estar à escuta na fila, no entanto, apenas um consumidor vai ler uma dada mensagem. Um fator importante neste tipo de modelos é o ***Load Balancing***[2]. O *load balancing* (balanceamento de carga) permite distribuir carga de forma aproximadamente igual entre os "trabalhadores". Neste caso, a carga corresponde à leitura e processamento das mensagens. Algumas *frameworks* suportam de raiz um mecanismo simples de balanceamento de carga [Marcos, 2016]. Este mecanismo vai registando os consumidores que já efetuaram trabalho e apenas permite a consumidores menos sobrecarregados ler uma nova mensagem, na tentativa de assegurar uma melhor gestão dos recursos e melhor desempenho (assumindo uma capacidade de processamento aproximadamente igual entre os diferentes consumidores).

Após a leitura de uma mensagem, um consumidor deve confirmar a receção da mesma através de um sinal [Marcos, 2016]. Quando chega o sinal de confirmação para que se saiba que a mensagem foi realmente entregue, esta é eliminada da fila.

Estes mecanismos podem depois ser aplicados de duas formas diferentes [Celar et al., 2016]:

- ***Peer-to-Peer***, em que a coordenação de descoberta e interação com outros nós é responsabilidade de cada nó.
- **Arquitetura com *Broker***, em que toda a comunicação é feita com uma entidade central que coordena a descoberta e interação com os restante nós.

Os padrões de interação existentes permitem comunicação simples entre apenas duas entidades seguindo o modelo *Point-to-Point*, mas também comunicação entre múltiplas entidades seguindo o modelo *Publish/Subscribe*. No modelo *Point-to-Point* existem duas entidades principais, o emissor e o recetor, que partilham uma fila para onde o emissor envia mensagens e de onde o recetor as lê [Ahuja and Mupparaju, 2014].

2.4.1 Modelo *Publish/Subscribe*

Em ***Publish/Subscribe***, que é "nos dias de hoje, o método mais utilizado para difusão de informação"[Marcos, 2016], há um sistema de subscrições por parte dos elementos da rede, e os subscritores recebem mensagens que correspondem à sua subscrição. Esta subscrição é baseada em parâmetros das mensagens, que permitem o redireccionamento para um destinatário que esteja interessado na mensagem, podendo ser feita de acordo com vários modelos:

- ***Topic-based Model:*** Neste modelo, cada mensagem/evento é marcada com o identificador do tópico em que é publicada. Os subscritores revelam interesse num conjunto de tópicos através de subscrição. Um tópico pode ser visto como um canal virtual que liga produtores e consumidores [Querzoni, 2012]. É responsabilidade dos produtores definirem os tópicos aos quais os consumidores podem subscrever [Chen et al., 2018]. O consumidor é responsável pela filtragem das mensagens, que sendo do tópico subscrito, podem não ser do seu

[2]Mais informação disponível em https://www.nginx.com/resources/glossary/load-balancing/

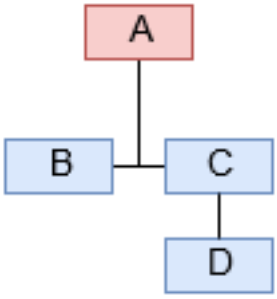


Figura 2.3: *Hierarchy-based Model*

interesse, provocando assim mais sobrecarga no recetor. Este é o modelo mais utilizado por plataformas orientadas a mensagens [Marcos, 2016].

- ***Hierarchy-based Model:*** Neste modelo, tal como no anterior, cada mensagem/evento é marcada com o identificador do tópico em que é publicada. No entanto, neste modelo os subscritores para além de receberem as mensagens dos tópicos que subscreveram, recebem também as mensagens pertencentes a tópicos da subárvore dos tópicos subscritos. Assim, numa hierarquia como a da Fig. 2.3, um consumidor que tenha subscrito o tópico A recebe as mensagens pertencentes aos tópicos A, B, C, e D. Este modelo foi criado na tentativa de colmatar alguma da falta de opções de filtragem das mensagens recebidas que o modelo *Topic-Based* padrão apresenta [Shen et al., 2010].

- ***Filtered-Topic Model:*** Neste modelo, tal como no Topic-Based, cada mensagem/evento é marcada com o identificador do tópico em que é publicada. No entanto, é aplicada mais uma fase de filtragem, de acordo com as preferências do consumidor, quando a mensagem é recebida. Mensagens que não cumpram os requisitos do consumidor não são entregues ao nível da aplicação, sendo descartadas [Corsaro et al., 2006]. Este modelo é também uma tentativa de colmatar alguma da falta de opções de filtragem das mensagens recebidas que o modelo *Topic-Based* padrão apresenta [Shen et al., 2010].

- ***Content-based Model:*** Neste modelo, o consumidor pode receber as mensagens nas quais tem interesse, definindo, na subscrição, um conjunto de restrições compostas por conjunções e disjunções [Corsaro et al., 2006]. A grande diferença para o modelo *Filtered-Topic* é que a filtragem, usando as restrições do utilizador, não precisa de ser feita no consumidor, pode ser feita em qualquer ponto do sistema [Shen et al., 2010]. Este modelo permite uma filtragem mais seletiva e expressiva quanto aos interesses do consumidor, não sendo preciso conhecer os nomes e conteúdos dos tópicos antes de subscrever [Shen et al., 2010]. No entanto, a maior expressividade provoca também maior *overhead* no cálculo dos consumidores interessados na mensagem [Corsaro et al., 2006].

- ***Type-based Model:*** Neste modelo, a subscrição é baseada em tipos que podem encapsular atributos e métodos [Shen et al., 2010]. Uma mensagem ou evento será, neste caso, um objeto. Este modelo permite um grau de filtragem intermédio, entre *Content-based* e *Topic-based*, porque utiliza a estrutura do *Topic-based* e permite a maior seletividade do *Content-based*, através do uso de restrições sobre os atributos do objeto [Shen et al., 2010].

2.4.2 Protocolos para *Middleware* Orientados a Mensagens

Ao longo dos anos vários protocolos para *middleware* orientados a mensagens foram surgindo, sendo de seguida descritos [Celar et al., 2016, Szydlo et al., 2017].

- ***Java Message Service (JMS)***, desenvolvido pela comunidade do Java, permite a troca de mensagens de forma assíncrona e com diferentes níveis de Qualidade de Serviço (QoS, do inglês Quality of Service). Este protocolo tem suporte para diversos modelos de partilha como *Publish-Subscribe* e *Point-to-Point*, usando um *broker* [Mahmoud, 2004]. Algumas implementações deste protocolo são o OpenMQ[3] e o TIBCO Enterprise Message Service[4].

- ***Advanced Message Queuing Protocol (AMQP)*** pode ser dividido em modelo de transporte e modelo de enfileiramento [Celar et al., 2016]. O modelo de enfileiramento consiste em dois principais conceitos: as filas e os *exchanges*. Neste arquitetura, usando um broker, um produtor envia uma mensagem para o *broker*, especificando o *exchange* para onde quer enviar a mensagem e uma *routing key*. Podemos pensar no *exchange* como um posto de correios, em que uma mensagem enviada para o *exchange* é depois redirecionada para a fila de destino correta. Uma *routing key* é um atributo da mensagem, analisado para a decisão sobre para que fila ou filas de destino enviar a mensagem [Aiyagari et al., 2008]. A seleção das filas de destino baseia-se também no tipo de *exchange*. Os tipos de *exchanges* que existem são:

  - **Direct**, em que uma mensagem que chegue ao *exchange* é enviada para as filas cuja *routing key* seja igual à *routing key* da mensagem. Para mensagens com correspondência para várias filas, são criadas e entregues cópias a cada uma delas. As mensagens que não tenham correspondência com nenhuma fila são automaticamente descartadas.
  - **Fanout**, em que uma mensagem que chegue ao *exchange* é enviada para todas as filas ligadas ao *exchange* [Aiyagari et al., 2008]. Ideal quando se pretende informar todos os nós da rede, como num *broadcast* [Marcos, 2016].
  - **Topic**, em que uma mensagem que chegue ao *exchange* é enviada para as filas cujo *routing pattern* faça correspondência com a *routing key* da mensagem. A diferença entre este tipo de *exchange* e o *direct* é a possibilidade da *routing key* das filas de mensagens ser uma expressão regular que faça correspondência com a *routing key* da

[3] Mais informação disponível em https://javaee.github.io/openmq/Overview.html
[4] Mais informação disponível em https://www.tibco.com/products/tibco-enterprise-message-service

mensagem. Esta *routing key* pode ter dois carateres especiais (o "*"e o "#"), dando origem ao *routing pattern* [Aiyagari et al., 2008, Marcos, 2016].

– **Headers**, em que a mensagem que chegue ao *exchange* é enviada para as filas baseado nas correspondências entre os *headers* da mensagem e o *binding* associado à fila. Para este *exchange* surge um parâmetro especial nos *headers*, "x-match", que pode ter associado o valor "all", para o qual todos os *headers* têm de fazer correspondência com o *binding*, ou o valor "any"em que basta pelo menos um dos *headers* fazer correspondência [Aiyagari et al., 2008].

Várias plataformas implementam o protocolo AMQP, como o RabbitMQ e o ActiveMQ.

- ***RESTful Messaging Service (RestMS)*** permite o envio de mensagens com formato XML (*Extensible Markup Language*), JSON (*JavaScript Object Notation*) e um conjunto de tipos MIME[5] (*Multipurpose Internet Mail Extensions*), usando o protocolo HTTP/HTTPS [RestMS, 2008]. O protocolo foi desenvolvido tendo como base o protocolo AMQP, com o objetivo de fornecer a interoperabilidade típica deste a aplicações Web. Assim, este protocolo tenta criar uma especificação que permita o mapeamento das propriedades do AMQP usando o conceito REST[6] (*Representational State Transfer*), que lhe permite fazer a indicação das filas/URLs para onde as mensagens são enviadas [Hintjens, 2008].

- ***Extensible Messaging and Presence Protocol (XMPP)*** permite de uma forma genérica trocar mensagens no formato XML. Foi desenvolvido principalmente para a troca de mensagens instantâneas, chamadas de voz e vídeo [XMPP, 2016]. Uma das suas caraterísticas diferenciadoras é a notificação de presença, isto é, a capacidade de fornecer dados dos intervenientes em tempo real indicando se um determinado cliente está ou não disponível [Saint-Andre, 2020]. Esta tecnologia usa um servidor, como o *broker* de AMQP, apresentando também carateristicas que lhe permitem fazer parte desta lista de MOM como a capacidade *publish-subscribe* assegurando QoS e segurança na comunicação. Existem várias implementações deste protocolo (clientes, servidores e bibliotecas) [XMPP, 2020].

- ***Streaming Text Oriented Messaging Protocol (STOMP)*** foi desenhado com o objetivo de ser mais simples e interoperável quando comparado com outros protocolos [STOMP, 2012a]. Neste protocolo uma mensagem é constituída por um comando, um conjunto de *headers* opcionais e o conteúdo da mensagem (ou seja, semelhante a um pedido HTTP). Este protocolo utiliza um servidor/*broker* para onde as mensagens são enviadas, mas não especifica a forma como as mensagens são depois encaminhadas para os destinatários. Existem várias implementações deste protocolo (clientes e servidores) [STOMP, 2012b].

- ***Message Queue Telemetry Transport (MQTT)*** foi desenhado para troca de mensagens em dispositivos de baixa capacidade, como dispositivos movéis ou IoT, sendo bastante mais

[5]Mais informação disponível em https://developer.mozilla.org/pt-BR/docs/Web/HTTP/Basico_sobre_HTTP/MIME_types

[6]Mais informação disponível em https://www.codecademy.com/articles/what-is-rest

eficiente energeticamente quando comparado com outros como o AMQP [Gilchrist, 2016]. Este protocolo é também bastante apropriado para ambientes com limitações de largura de banda para a comunicação, mas menos apropriado para sistemas complexos de troca de mensagens, quando comparado com outros protocolos como o AMQP [Luzuriaga et al., 2015a]. Este protocolo usa uma arquitetura *publish-subscribe* com *broker* para a troca de mensagens [Liu et al., 2020]. Um dos seus casos de uso típicos seria a monitorização e envio periódico de leituras de sensores, associados a um número elevado de máquinas de baixo custo que utilizam MQTT para reportar as leituras a um servidor/*broker* central [Gilchrist, 2016]. Existem várias servidores/*brokers* para este protocolo como o HiveMQ[7] e o Mosquitto[8] [Celar et al., 2016].

- ***Data Distribution Service (DDS)*** é desenvolvido pelo Object Management Group (OMG[9]) e foi desenhado para sistemas de tempo real. Este protocolo segue uma arquitetura centrada nos dados (*data-centric*), que apenas usa mensagens para trocar os dados entre os diferentes nós da rede. Este algoritmo segue o modelo *Data-Centric Publish-Subscribe* (DCPS) em que a comunicação corresponde à leitura e escrita usando um espaço de dados global [Object Management Group, 2015]. Este espaço de dados é, no entanto, virtual porque a maioria das implementações de DDS usa comunicação *peer-to-peer* entre produtores e consumidores (isto é, sem *broker*) [Stan Schneider, 2013]. Este modelo permite que os vários nós da rede, interessados num determinado tópico, tenham dados sobre esse tópico sincronizados em tempo real, com cada nó a servir de produtor e/ou consumidor. Assim, esta arquitetura é mais orientada à sincronização de dados entre múltiplos nós, principalmente num estilo muitos-para-muitos, do que à comunicação baseada em mensagens entre dois ou mais nós [Stan Schneider, 2013]. A caraterística diferenciadora do DDS é o conjunto vasto de políticas de QoS que apresenta em conjunto com o protocolo UDP (*User Datagram Protocol*), permitindo garantir fiabilidade mas mantendo um desempenho superior, quando comparado com o uso do protocolo TCP (*Transmission Control Protocol*) [Object Management Group, 2015]. Este protocolo fornece ainda um sistema que permite a descoberta dinâmica de consumidores interessados num tópico, usando *multicast* [Hans Van't Hag, 2017]. Existem várias implementações não comerciais deste protocolo como o OpenDDS[10], o Vortex OpenSplice Community Edition[11] e o Eclipse Cyclone DDS[12].

Para além destes protocolos existem alguns menos usados e referenciados como: o protocolo proprietário *Microsoft Message Queue* (MSMQ[13]) desenvolvido pela Microsoft; e o *Open Middleware Agnostic Messaging API* (OpenMAMA[14]) desenvolvido com o intuito de fornecer uma

[7] Mais informação disponível em https://www.hivemq.com/hivemq/
[8] Mais informação disponível em https://mosquitto.org/
[9] Mais informação disponível em https://www.omg.org/about/index.htm
[10] Mais informação disponível em https://opendds.org/
[11] Mais informação disponível em https://github.com/ADLINK-IST/opensplice
[12] Mais informação disponível em https://projects.eclipse.org/projects/iot.cyclonedds
[13] Mais informação disponível em https://docs.microsoft.com/en-us/previous-versions/windows/it-pro/windows-server-2008-R2-and-2008/cc726051(v%3dws.10)
[14] Mais informação disponível em http://www.openmama.org/what-openmama

API que permite interagir de forma uniforme com vários *middleware* diferentes. Algumas implementações mais especificas e que não seguem diretamente nenhum destes protocolos, como o Apache Kafka e o ZeroMQ são estudadas em detalhe na secção 3.2. Também na secção 3.2 são estudados o RabbitMQ e o ActiveMQ, plataformas poliglotas que permitem o uso de vários dos protocolos referidos acima.

2.5 Segurança na comunicação

Devido a uma arquitetura descentralizada e distribuída, os sistemas multi-agente apresentam grandes desafios no que diz respeito à comunicação segura. A identidade de um agente, por exemplo, torna-se muito complicada e desafiante de garantir sem a existência de uma autoridade centralizada que o certifique [Heydari and Effatparvar, 2013, Dorri et al., 2018].

O ambiente em que uma plataforma multi-agente está inserido influencia as vulnerabilidades a que os agentes estão expostos. Num ambiente controlado e isolado, em que as motivações são conhecidas e fundamentalmente os agentes podem confiar uns nos outros, as vulnerabilidades de segurança são pouco preocupantes. No entanto, para um uso num ambiente aberto, com um conjunto de agentes que não se tem a garantia de serem confiáveis, possivelmente com agentes noutras plataformas que comunicam pela internet, várias vulnerabilidades podem ser exploradas [Heydari and Effatparvar, 2013, Dorri et al., 2018]:

- A comunicação pode ser intercetada, possibilitando ao atacante monitorizar ou até mesmo alterar o conteúdo de uma mensagem se não forem usados mecanismos de segurança adequados. Este tipo de ataque é conhecido por *man-in-the-middle*, podendo ser visto na Fig. 2.4. Na figura é possível ver a comunicação entre dois veículos (a Alice e o Bob) a ser intercetado por um terceiro elemento malicioso (a Eve).

- Os agentes como entidades autónomas, que interagem, cooperando ou competindo num dado ambiente, reagem de acordo com a informação que recebem do ambiente em que estão inseridos, nomeadamente mensagens de outros agentes. Assim um agente malicioso pode, aproveitando-se da reatividade típica dos agentes, tentar interferir com o processo de decisão de um outro agente através da partilha de informações falsas.

- A mobilidade dos agentes entre plataformas pode levar a que um agente, a que tenham sido dadas informações falsas por um outro agente malicioso (por exemplo através de mensagens alteradas), propague informações falsas a outros agentes das plataformas onde passa.

Assim, para garantir o funcionamento esperado e mitigar os riscos de segurança, existem alguns requisitos que um sistema multi-agente deve assegurar [Borselius, 2002, Dorri et al., 2018]:

- **Integridade** garantindo que a mensagem recebida é de facto a mensagem enviada, isto é, não houve alteração do seu conteúdo.
- **Confidencialidade** garantindo que o conteúdo da mensagem é cifrado, para que apenas os agentes autorizados (i.e., que possuem a chave) consigam perceber o seu conteúdo.

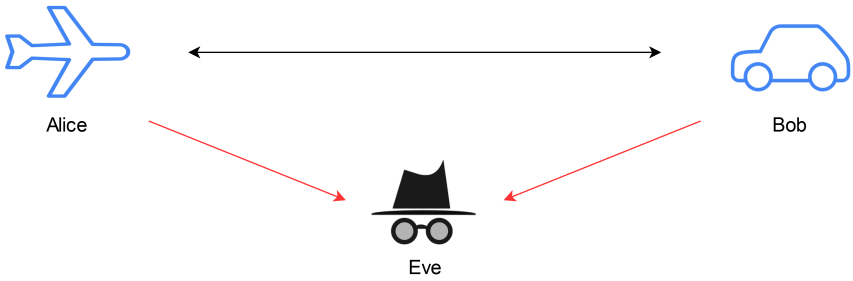


Figura 2.4: Ataque *Man-in-the-middle*

- **Autenticidade** garantindo a identidade de um agente. Na troca de mensagens permite garantir que a mensagem recebida foi de facto enviada por quem diz ser autor da mesma.
- **Autorização** garantindo que um agente apenas consegue aceder ao conteúdo para o qual tem permissão.
- **Disponibilidade** garantindo que os serviços e recursos estão disponíveis para agentes autenticados e autorizados, sempre que pedidos.
- ***Non-Repudiation*** garantindo que um agente não pode negar ter enviado uma mensagem que realmente enviou.

Estes requisitos podem ser assegurados usando o conceito de cifragem. A cifragem utiliza algoritmos para ocultar o significado de uma mensagem [Denis and Johnson, 2007]. Nas secções seguintes são detalhadas a cifragem unidirecional e a cifragem bidirecional.

2.5.1 Cifragem Unidirecional

A cifragem unidirecional consiste num processo em que uma dada entrada (*input*) é transformada numa saída (*output*), de tal forma que esta tranformação/mapeamento seja impossível de reverter [Denis and Johnson, 2007]. Este tipo de cifragem é normalmente utilizado para assegurar integridade, através do uso de *hashing* criptográfico.

As funções de *hashing* criptográfico permitem criar um valor, chamado *digest*, com tamanho fixo a partir de um *input* de tamanho variável. Este tipo de funções devem garantir algumas propriedades [Ning, 2011]:

- **Tamanho de *Output* Fixo**, isto é, a produção de um *output* de tamanho fixo.
- **Resistência à pré-imagem**, garantindo que dado um determinado *hash* h é computacionalmente inviável encontrar a mensagem M que lhe deu origem.
- **Resistência à segunda pré-imagem**, garantindo que tendo uma mensagem M1 é computacionalmente inviável encontrar uma outra mensagem M2 com o mesmo *hash*.
- **Desempenho**, garantindo que é computacionalmente eficiente calcular o *hash* de uma dada mensagem.

Existem várias funções de *hashing* criptográfico como SHA-256, SHA-384, SHA-512, SHA-3, entre outros [Haque et al., 2018].

2.5.2 Cifragem Bidirecional

A cifragem bidirecional tem como objetivo cifrar um conteúdo (*input*) com o uso de uma chave, de tal forma que seja possível reverter o efeito da cifragem. O conteúdo original pode depois ser recuperado através do uso dessa mesma chave ou de uma outra chave [Jackson, 2013].

2.5.2.1 Encriptação de Chave Privada

Este tipo de encriptação, também conhecida por encriptação de chave simétrica, utiliza um sistema de chave partilhada que precisa de ser acordada entre os intervenientes antes do estabelecimento da comunicação. Esta chave é utilizada para ambos o processo de encriptação e o processo de desencriptação do conteúdo da mensagem.

A principal garantia que este sistema nos oferece é a confidencialidade, assumindo que a chave se mantém secreta. Isto implica uma troca inicial de forma segura usando um outro algoritmo.

Assim, com este tipo de sistema, a gestão das chaves torna-se bastante complicada. Assumindo que um agente utiliza uma chave diferente para a comunicação com cada outro agente, isso significa que necessitaria de fazer a gestão de N chaves para a comunicação com N outros agentes. Dessa forma, seria, no entanto, possível garantir autenticidade porque o outro agente seria o único para além do próprio a ter conhecimento da chave. A assinatura da mensagem podia também ser gerada através da encriptação do *hash* da mensagem com a chave partilhada. A garantia de *non-repudiation* é que seria impossível de obter com este tipo de encriptação, porque, dados dois agentes A e B num ambiente X, A não consegue garantir, perante todos os agentes de X, que foi B que lhe enviou uma determinada mensagem, dado que ambos conhecem a chave e ambos poderiam assinar a mensagem.

Existem várias implementações deste sistema como AES, DES, 3DES, Blowfish, entre outros [Haque et al., 2018].

2.5.2.2 Encriptação de Chave Pública

Este tipo de encriptação, também conhecida por encriptação de chaves assimétricas, permite que duas entidades, sem conhecimento ou acordo prévio, comuniquem com algumas garantias de segurança, nomeadamente: integridade, confidencialidade, autenticidade e *non-repudiation*. Neste tipo de encriptação existem duas chaves, a chave privada e a chave pública. Estas chaves, associadas a uma entidade, são geradas usando alguns algoritmos criptográficos baseados em algumas propriedades e provas matemáticas. A chave pública pode ser conhecida por qualquer entidade, inclusive um possível atacante. Quanto à chave privada, é um dos pressupostos do algoritmo que esta se deve manter secreta, sendo conhecida apenas por quem a gerou. Estas duas chaves funcionam de forma que uma consegue inverter o efeito da outra, ou seja, o que a chave pública encriptar apenas a chave privada consegue desencriptar e vice-versa [CloudFlare, 2020].

A forma como as chaves são utilizadas oferece diferentes garantias de segurança, nomeadamente [Mollin, 2002]:

- **Confidencialidade:** O emissor de uma mensagem pode garantir que apenas o recetor consegue perceber o conteúdo da mensagem, se a encriptar com a chave pública do recetor. O uso da chave pública do recetor para a encriptação garante-nos que apenas a chave privada do recetor (que se assume ser secreta) conseguirá desencriptar o seu conteúdo.

- **Autenticidade:** O emissor de uma mensagem pode garantir ao recetor que foi realmente ele o emissor daquela mensagem, se a encriptar com a sua chave privada e o recetor a desencriptar com sucesso usando a chave pública do emissor.

- **Integridade:** A integridade de uma mensagem é normalmente garantida através da concatenação, à mensagem original, do seu *hash* (**<mensagemM><*hash*-da-mensagemM>**). Este tipo de garantia foi desenvolvido inicialmente para a existência de ruído na comunicação, permitindo a deteção de alterações ao conteúdo da mensagem durante o transporte. No entanto, numa perspetiva de segurança apenas com o uso do *hash*, um atacante pode alterar o conteúdo da mensagem e refazer também o *hash*. Por isso é necessário assinar o *hash* da mensagem, surgindo o conceito de assinatura digital. Com o uso da assinatura digital o atacante não consegue assinar o novo hash (da nova mensagem, falsa) de forma correta e imperceptível ao recetor, porque não conhece a chave privada do emissor. O emissor deve então assinar o *hash* da mensagem com a sua chave privada, garantindo que apenas ele o conseguiria fazer, e depois concatenar a assinatura à mensagem. Do outro lado, o recetor deve desencriptar o *hash* da mensagem usando a chave pública do emissor e comparar o resultado com o *hash* da mensagem recebida, que ele próprio deve calcular. Se forem iguais a mensagem não foi alterada.

- **Non-Repudiation:** O emissor de uma mensagem não pode negar ter sido o seu emissor, dado que usou a sua chave privada para a encriptar. Qualquer entidade consegue verificá-lo usando a chave pública desse emissor.

Tal como descrito, este mecanismo permite assegurar cada uma das garantias de segurança em separado. No entanto, é possível assegurar o seu conjunto usando um processo completo como o descrito na Fig. 2.5.

Neste esquema temos duas entidades a comunicar, a Alice e o Bob. A Alice, para enviar uma mensagem ao Bob, precisa inicialmente de gerar o *hash* da mensagem a enviar. Este *hash* é depois encriptado com a chave privada da Alice, isto é, assinado. A mensagem que é enviada ao Bob é, na realidade, um bloco (mensagem concatenada do seu *hash*) que será totalmente encriptado agora usando a chave pública do Bob. Na receção da mensagem, o Bob começa por desencriptar o pacote com a sua chave privada. De seguida, utiliza a chave pública da Alice para desencriptar o *hash* recebido, sendo este *hash* comparado com o valor do *hash* que o Bob calcula para a mensagem efetivamente recebida. Se ambos tiverem o mesmo valor, significa que a mensagem não

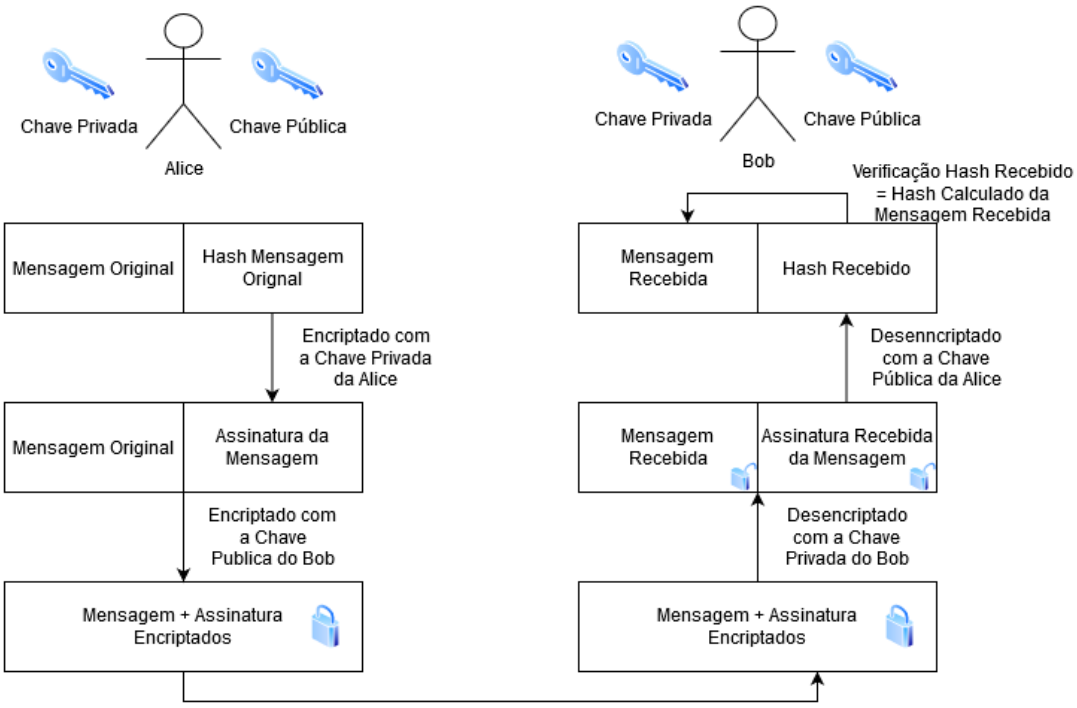


Figura 2.5: Encriptação de Chave Pública Completa

foi alterada e o Bob processa-a normalmente. Caso contrário, o Bob sabe que a mensagem foi alterada durante o envio.

A chave pública apesar de ser partilhada com todos, é responsável direta ou indiretamente por algumas das garantias que foram sendo mencionadas, sendo uma parte fundamental do processo. Como não há uma associação entre um agente e a sua chave pública definida, surge uma vulnerabilidade. Como ter a certeza de que estamos a falar com quem diz estar do outro lado? Estes mecanismos podem, então, ser contornados por um terceiro elemento através de um ataque chamado *man-in-the-middle*. Assumindo a comunicação entre duas entidades legítimas, A e B, este terceiro elemento malicioso, C, pode utilizando o seu par de chaves estabelecer comunicação entre A e ele próprio e depois entre ele e B, fazendo com que A e B pensem estar a falar um com o outro. A solução encontrada para resolver esta vulnerabilidade passa pela existência de autoridades centralizadas, em que se confia. Estas autoridades certificam que uma determinada chave pública pertence realmente a uma determinada entidade através da emissão de um certificado[15]. Este certificado, normalmente chamado certificado digital, corresponde à chave pública da entidade que se pretende certificar assinada pela autoridade centralizada com a sua chave privada, associada a um identificador dessa entidade. Este certificado permite depois verificar, usando a chave pública da autoridade que o emitiu (confiável e conhecida por todos), que aquela chave pública pertence àquela entidade [Henmi, 2006].

[15]Mais informação disponível em https://www.venafi.com/education-center/pki/how-does-pki-work

Existem várias implementações deste sistema como DSA, ElGamal, RSA, criptografia de curva elíptica, entre outros [Haque et al., 2018].

A criptografia de curva elíptica, descrita no anexo A.1, apresenta uma grande vantagem em relação aos restantes algoritmos. Este tipo de criptografia, baseada na estrutura algébrica de curvas elípticas sobre corpos finitos, consegue manter um mesmo nível de segurança com tamanhos de chaves muito inferiores, quando comparada, por exemplo, com o algoritmo RSA (descrito em A.2), tornando-se mais eficiente e rápida de computar. Como referido em [Sullivan, 2013], quebrar uma chave RSA de 228 bits requer menos energia que ferver uma colher de chá de água, mas quebrar uma chave de curva elíptica de 228 bits requer energia suficiente para ferver toda a água do planeta Terra.

2.5.2.3 Comparação entre Encriptação de Chave Pública e Encriptação de Chave Privada

A encriptação de chave pública apresenta algumas vantagens quando comparada com encriptação de chave privada, nomeadamente [Mollin, 2002]: tempo de vida útil das chaves é bastante mais longo; a gestão das chaves para um ambiente de comunicação com muitos intervenientes é mais simples, em contraste com o armazenamento de uma chave por cada um dos intervenientes na encriptação de chave privada; não necessidade de troca de chaves anterior à comunicação entre entidades que não se conhecem (troca de chaves vista em mais detalhe no anexo B).

No entanto, apresenta também algumas desvantagens, nomeadamente: computação bastante menos eficiente e por isso mais lenta (segundo [Haque et al., 2018], em média, quase 1000 vezes mais lenta); tamanho de chaves necessárias para o mesmo grau de segurança são bastante superiores mesmo quando comparado com criptografia de curva elíptica (segundo [Mollin, 2002], em média, quase 10 vezes maiores).

2.5.3 *Transport Layer Security*

O *Transport Layer Security* (TLS) é um protocolo normalmente usado para comunicação segura entre 2 entidades [Koschuch et al., 2012]. Este protocolo usa um mecanismo híbrido com encriptação de chave privada e encriptação de chave pública. O protocolo funciona sob a forma de sessões que podem ser estabelecidas entre 2 entidades. No inicio de uma sessão, é usado o mecanismo de encriptação de chave pública para permitir autenticação de uma ou ambas as partes e para permitir a troca de uma chave secreta (apenas as duas entidades conhecem a chave mesmo que todas as mensagens tenham sido intercetadas por elementos maliciosos). Depois, durante o resto da sessão essa chave é usada, juntamente com o mecanismo de chave privada, para que a comunicação aconteça de forma segura, possibilitando comunicação mais eficiente, rápida e segura.

Este protocolo teve origem num outro protocolo chamado *Secure Socket Layer* (SSL) e está neste momento na versão 1.3 (lançada em agosto 2018) [Rescorla, 2018]. Neste momento todas as versões do protocolo SSL são consideradas inseguras, bem como as versões 1.0 e 1.1 do TLS [Barnes et al., 2015, Moriarty and Farrell, 2020]. Assim, devem ser usadas, pois são consideradas seguras, as versões 1.2 e 1.3 do TLS.

Capítulo 3

Estado da Arte

Este capítulo fornece um estado da arte em alguns tópicos importantes para esta dissertação. Está dividido em dois grupos principais - plataformas multi-agente e plataformas para *Middleware* Orientados a Mensagens.

3.1 Plataformas Multi-Agente

Foi feita uma análise das plataformas multi-agente existentes no mercado, de forma a perceber quais as suas características, principalmente no que diz respeito aos *middleware* de comunicação que utilizam e se estas vão de encontro ao que se pretende com os objetivos desta dissertação. A análise revelou um elevado número de plataformas, maioritariamente desenvolvidas em Java e com desenvolvimento inicial há já bastante tempo. Assim, bastantes destas já não têm desenvolvimento ou suporte ativo, como o próprio AgentService. O AgentService, por se tratar do sistema multi-agente utilizado na plataforma do LIACC, foi de qualquer forma analisado em detalhe.

De forma a manter o suporte para as aplicações/agentes da plataforma do LIACC procuramos por plataformas que, tal como o AgentService, tivessem suporte para agentes externos desenvolvidos em C#. No entanto, não encontramos na literatura nenhuma outra plataforma com suporte para agentes externos em C# ou noutra linguagem, ou pelo menos que o referenciem como tal.

Decidimos procurar por plataformas que pelo menos permitissem o desenvolvimento de agentes *in-platform* em C#. Encontramos duas plataformas, o ActressMAS[1] e o UMAP[2] (Universal Multi-Agent Platform for .NET developers). O ActressMAS apresenta pouca documentação sobre a descrição do seu funcionamento, fazendo apenas uma enumeração dos protocolos e algoritmos que implementa. Na sua página Web é também disponibilizada alguma documentação específica do código fonte. Depois da análise do seu código fonte percebemos, que tal como o AgentService, o ActressMAS utiliza um mecanismo de *polling* para a receção de mensagens. Na literatura não encontramos projetos que referenciem usar esta plataforma. O UMAP (definido em [Mrozek et al., 2014b]) é referenciado como utilizado em [Mrozek et al., 2014a] e [Fatunmbi et al., 2017] mas

[1]Mais informação disponível em http://florinleon.byethost24.com/actressmas/
[2]Mais informação disponível em http://zti.polsl.pl/w3/dmrozek/umap.htm

não tem, aparentemente, suporte ativo e a documentação é quase inexistente (apenas encontramos informação na sua página Web mas muito pouca). Por estes motivos decidimos que ambas as plataformas não seriam um boa solução para os objetivos desta dissertação.

Decidimos focar-nos em plataforma mais referenciadas e adotadas pela comunidade, que apresentam desenvolvimento e suporte ativo recente e sigam as normas FIPA. Assim, o JADE (Java Agent DEvelopment Framework) foi uma das plataformas escolhidas, por ser uma das mais referenciadas ao longo do anos e com mais de 100 referências no Web of Science[3] nos últimos 3 anos. Para além disso, procuramos por plataformas que apresentassem um *middleware* orientado a mensagens, levando à escolha do JIAC (Java-based Intelligent Agent Componentware) e do SPADE (Smart Python Agent Development Environment). Espera-se que estas possam ser uma mais valia para esta dissertação, sendo apresentadas abaixo em detalhe. Estas plataformas podem, numa fase posterior desta dissertação, ser adaptadas para permitirem a capacidade de desenvolvimento de agentes externos em C#.

Estudos detalhados de outras plataformas podem ser encontrados em [Kravari and Bassiliades, 2015] e [Leon et al., 2015].

3.1.1 JIAC

O **JIAC** é uma *framework* e arquitetura multi-agente de código aberto, desenvolvida em Java e distribuída pelo DAI-Labor [DAI-Labor, 2020]. Esta *framework* permite e facilita o desenvolvimento e operação de serviços e aplicações distribuídas de larga escala, sendo considerada uma *framework* que cumpre as especificações FIPA [Soklabi et al., 2013]. A sua arquitetura baseia-se num conjunto de nós que formam a plataforma, comunicando entre si através de mensagens e serviços. Cada nó corre na sua *Java virtual machine* (JVM), sendo constituído por agentes. Cada agente corre depois no seu *thread* individualmente [DAI-Labor, 2019]. Esta *framework* suporta as várias fases de desenvolvimento de um sistema multi-agente, desde a conceção/design, passando pela implementação até ao *deployment*. Os principais objetivos do JIAC são a distribuição, escalabilidade, adaptabilidade e autonomia.

A comunicação entre 2 agentes JIAC é desde a versão 5 baseada em Apache ActiveMQ usando o protocolo JMS 1.1. O uso deste *middleware* de comunicação permite ao JIAC comunicar com agentes no mesmo computador, mas também com agentes noutros computadores na rede. Permite também suporte para *buffering* de mensagens quando um agente não está disponível, bem como reconexão depois de falhas na rede [Lützenberger et al., 2015].

Devido à sua arquitetura, o JIAC permite a distribuição dos agentes pela rede, isto é, os agentes podem correr em diferentes computadores na rede, em diferentes nós, mas todos pertencentes à mesma plataforma.

Esta *framework* suporta, através do uso do ActiveMQ, SSL/TLS (Secure Socket Layer/Transport Layer Security) usando certificados [DAI-Labor, 2017b]. Assim, permite assegurar confidencialidade, integridade e autenticidade na troca de mensagens [Freier et al., 2011]. Segundo

[3]Mais informação disponível em `https://apps.webofknowledge.com`

[Lützenberger et al., 2015], também a comunicação com o DF e o AMS utiliza o ActiveMQ, pelo que se espera que esta aconteça de forma segura.

A sua documentação mantém-se atualizada de acordo com a sua última versão, descrevendo bastante bem a arquitetura da plataforma e os seus componentes [DAI-Labor, 2017a]. A sua comunidade de suporte[4], apesar de não apresentar um elevado número de questões (apenas 8 questões em 2019), apresenta respostas e soluções rápidas para as questões apresentadas.

3.1.2 JADE

O **JADE** é uma *framework* de código aberto para sistemas multi-agente desenvolvida em Java e distribuída pela Telecom Itália [Telecom IT, 2017]. Esta *framework* permite simplificar a implementação de sistemas multi-agente através de um *middleware* que cumpre as especificações FIPA, bem como um conjunto de ferramentas gráficas que permitem e ajudam no *debugging* e *deployment* do sistema desenvolvido [Telecom IT, 2017]. O JADE permite ainda a distribuição por vários computadores, que podem ou não utilizar o mesmo sistema operativo, com a configuração a ser controlada remotamente usando uma interface gráfica. Os agentes podem ser ainda movidos entre computadores durante a execução. Em 2015, o JADE era a plataforma multi-agente mais popular no meio industrial e académico, que cumpre as especificações FIPA, sendo uma das mais referenciadas ao longo do anos [Kravari and Bassiliades, 2015]

A comunicação entre 2 agentes JADE acontece através de um protocolo do próprio JADE, o Internal Message Transport Protocol (IMTP). Este protocolo utiliza passagem de eventos do Java para comunicação dentro do mesmo *container* e *Remote Method Invocation* (RMI) para comunicação entre *containers* [Soklabi et al., 2013]. Em JADE um *container* representa um conjunto de agentes. Assim, os *containers* podem ser distribuídos pela rede, isto é, agentes podem correr em diferentes computadores na rede. No entanto, todos têm de se registar perante o *container* principal, *"Main Container"*, que é o primeiro a ser lançado em execução [Bellifemine et al., 2007].

Esta *framework* tem disponível através de um plugin, JADE-S, algumas funcionalidades de segurança, nomeadamente: autenticação, autorização para agentes executarem certas ações, integridade e confidencialidade na troca de mensagens [Vila et al., 2007]. Segundo a sua documentação, o acesso e realização de operações perante o AMS pode ser limitado, por operação, através do uso de permissões [JADE Board, 2005]. Para além disso, as mensagens são assinadas para garantir a identidade do emissor. No entanto, não é feita qualquer referência a que estes mecanismos estejam disponível para o uso do DF. Segundo [Moreno et al., 2003], para o DF apenas podem ser usadas permissões globais que permitem ou impedem o acesso total ao DF por um dado agente.

Esta *framework* tem disponível um plugin, JADE-LEAP, que lhe permite ser executada em dispositivos móveis que correm Java e em dipositivos que correm a .NET Framework [Caire and Pieri, 2011]. No entanto, a última versão deste plugin é de junho de 2012 enquanto que o JADE foi recebendo sucessivas atualizações até 2017. A última referência, no fórum de questões do

[4]Mais informação disponível em `http://jiac.de/qa/activity`

JADE, que encontramos associando ambos LEAP e .NET é de 2011 por um dos principais impulsionadores do projeto, Giovanni Caire, referindo que a associação entre ambos estava obsoleta e não recebia mais suporte [Caire, 2011]. A documentação refere ainda algumas limitações no uso deste plugin e encontramos também outras como a impossibilidade de utilização dos serviços de segurança que o JADE-S fornece.

A documentação referenciada no seu website é relativa já a versões lançadas em 2009/2010 (versões 3.7 e 4.0) enquanto que a versão mais atual foi lançada em 2017 (versão 4.5). A documentação existente é, no entanto, bastante detalhada na descrição do funcionamento da plataforma [Caire, 2009, Trucco et al., 2010]. A sua comunidade de suporte[5], apesar de não apresentar um elevado número de questões (apenas 25 questões em 2019), apresenta respostas e soluções rápidas para as questões apresentadas. No entanto, desde o último trimestre de 2019 que nenhuma mensagem surgiu. A subscrição do fórum, da qual a publicação de questões depende, não está neste momento a funcionar.

3.1.3 SPADE

O **SPADE** é uma plataforma de código aberto para desenvolvimento de sistemas multi-agente desenvolvida em Python [Palanca, 2019b]. A plataforma tem um conjunto de funcionalidades que ajudam no desenvolvimentos de MAS: um canal de comunicação; suporte FIPA usando *XMPP Data Forms* (baseado em XML); notificação de presença que permite saber o estado dos agentes em tempo real; e *Multi-user Conference* (MUC) que permite criar fóruns com grupos de agentes para atividades específicas (um leilão por exemplo) [Gregori et al., 2006, Palanca, 2019a].

Os agentes SPADE podem ser criados através de um módulo chamado *SPADE Agent Library* (biblioteca de agentes SPADE). Este módulo consiste num conjunto de classes, funções e ferramentas para criar agentes na plataforma SPADE. A comunicação entre dois agentes SPADE é baseada em mensagens instantâneas (*XMPP/Jabber technology*) [XMPP, 2016]. O XMPP é um protocolo de comunicação para *middleware* orientados a mensagens estudado em 2.4.2. Todos os componentes e agentes da plataforma estão ligados a um elemento central (servidor), o *XML Router*, que é responsável por receber uma mensagem e a redirecionar para o destinatário.

Esta *framework* fornece, de raiz, algumas funcionalidades que permitem uma maior segurança na comunicação dos seus agentes [Gregori et al., 2006]:

- A autenticação no *XML router* é feita com nome de utilizador e palavra-passe, assegurando que apenas agentes autorizados podem enviar mensagens.

- A conexão entre um agente e o *XML router* pode ser encriptada usando SSL/TLS, assegurando autenticidade (por parte do cliente e do *XML router*), integridade e confidencialidade nas mensagens trocadas.

[5] Mais informação disponível em `https://jade.tilab.com/pipermail/jade-develop/`

- O *XML router* preenche o campo que indica a origem da mensagem com a identidade do agente que realmente lhe enviou a mensagem, assegurando que um agente não consegue fazer-se passar por outro (ataque de roubo de identidade).

No entanto, não encontramos referências a que estes mecanismos de segurança possam ser aplicados na comunicação com o DF e o AMS.

A sua documentação mantém-se atualizada de acordo com a sua última versão, descrevendo de forma bastante intuitiva e com uso de exemplos os mecanismos da plataforma [Palanca, 2019c]. A sua comunidade de suporte[6], apesar de não apresentar um elevado número de questões (apenas 22 questões em 2019), apresenta respostas e soluções rápidas para as questões apresentadas.

3.1.4 AgentService

O **AgentService** é uma *framework* de código aberto para desenvolvimento de sistemas multi-agente desenvolvida em C# e baseada em *Common Language Infrastructure* (CLI) [Vecchiola et al., 2009]. As suas principais funcionalidades passam pela flexibilidade dos agentes e pela modularidade da plataforma. Assim, permite aos agentes verdadeiro *multi-tasking* e mudança de plataforma durante a execução. O sistema pode ser customizado usando várias ferramentas: *visual protocol designer* (designer de protocolo visual), um conjunto de extensões à linguagem e ferramentas para simplificar o controlo sobre o MAS [Vecchiola et al., 2008]. Foi desenvolvido na Universidade de Génova, mas desde setembro de 2009 que não existem desenvolvimentos na *framework*, coincidindo com o abandono do projeto por parte de um dos seus impulsionadores, Andrea Passadore [Silva, 2011]. O seu website[7] foi também descontinuado em 2017.

No AgentService, os agentes dentro da mesma plataforma podem comunicar através de várias tecnologias: um *Web Service* em que são utilizadas mensagens SOAP[8] (*Simple Object Access Protocol*) serializadas em formato XML; MSMQ com uma fila associada a cada agente; .NET Remoting[9], em que um agente expõe uma interface remota que os outros podem chamar para lhe enviar uma mensagem. No entanto, apenas o *.NET Remoting* e o *ASP.NET Web Services*[10] permitem a comunicação entre agentes em plataformas diferentes [Vecchiola et al., 2008]. Em relação a segurança estas tecnologias apresentam diferentes níveis: *.NET Remoting* não é recomendado que seja usado na internet, apenas em ambientes mais seguros e controlados como uma rede interna [Sotolár, 2013]; MSMQ e *ASP.NET Web Services* garantem confidencialidade, integridade e autenticidade [Breakwell, 2008a, Breakwell, 2008b, Santos, 2006].

Esta *framework* permite que aplicações externas possam ser integradas como agentes na plataforma, utilizando o *AgentService External Runtime* (AER) [The AgentService Team, 2008]. Estes

[6] Mais informação disponível em https://github.com/javipalanca/spade/issues

[7] Imagem arquivada disponível em https://web.archive.org/web/20170703015137/http://www.agentservice.it/

[8] Mais informação disponível em http://www4.di.uminho.pt/~jcr/LIVROS/EngWeb-OpenBook/ch01.html

[9] Mais informação disponível em https://docs.microsoft.com/en-us/openspecs/windows_protocols/ms-netod/bfd49902-36d7-4479-bf75-a2431bd99039

[10] Mais informação disponível em https://dotnet.microsoft.com/apps/aspnet

agentes são capazes de executar as mesmas funcionalidades que um agente da plataforma, nomeadamente, as funcionalidades mais importantes como: registar-se na plataforma, enviar e receber mensagens de outros agentes (externos ou não), aceder às páginas amarelas e páginas brancas. Para a comunicação com este tipo de agentes são utilizados os serviços WCF (*Windows Communication Foundation*) [The AgentService Team, 2008, Passadore et al., 2008], mas a receção de mensagens por um agente externo à plataforma é bastante ineficiente. É possível recorrer a *polling*, verificando com um determinado período se existem novas mensagens para serem lidas, o que para além de ineficiente, cria um atraso na receção da mensagem, que no máximo corresponde ao valor do período de *polling* utilizado. É também possível recorrer a um método bloqueante que nos retorna à primeira mensagem recebida, mas que utiliza também ele *busy waiting* internamente [The AgentService Team, 2008]. Os serviços WCF garantem confidencialidade, integridade e autenticidade através do uso de *bindings* [Chappell, 2005]. No entanto, a documentação do AgentService não faz qualquer referência à possibilidade de uso de comunicação segura na plataforma. Isto implica que a comunicação entre agentes, mas também a comunicação com o DF e o AMS, seja realizada de forma insegura.

Apesar de descontinuado, a documentação que conseguimos encontrar parece bastante detalhada, com a descrição das várias funcionalidades em manuais[11] e em documentação no próprio website[12]. Para além disso, tinham disponível um fórum[13] para esclarecimento de questões.

3.1.5 Resumo Comparativo das Plataformas Multi-Agente

A Tabela 3.1 mostra um resumo comparativo das principais caraterísticas das plataformas multi-agente estudadas em detalhe. De destacar que todas elas apresentam suporte FIPA, mas apenas o AgentService apresenta a capacidade de integração de agentes externos. Estas duas funcionalidades pretende-se que se mantenham mesmo que este sistema multi-agente seja substituído por um outro.

Estas plataformas foram já utilizadas para as mais diversas aplicações no mundo real, demonstrando a sua capacidade e o que permitem atingir. O JIAC já foi utilizado para diversos projetos como: *Energy Efficiency Controlling in the Automotive Industry* (EnEffCo), que tem como objetivo estabelecer técnicas para melhorar a produção em termos de custo e eficiência energética [Küster et al., 2013]; *Intelligent Solutions for Protecting Interdependent Critical Infrastructures* (ILIas), que permite a simulação de ataques (naturais ou feitos pelo homem) a infraestruturas e avalia o comportamento dos mecanismos de proteção [Konnerth et al., 2012]; *Service Centric Home* (SerCHo), uma plataforma personalizável para *smart home* [Hirsch et al., 2006]. O JADE é também utilizado em diversos sistemas, como: um sistema que permite a simulação da alocação e realização de tarefas de transporte e produção num sistema que controla veículos autoguiados na

[11] Imagem arquivada disponível em https://web.archive.org/web/20151002021213/http://www.agentservice.it/wiki/

[12] Imagem arquivada disponível em https://web.archive.org/web/20151002021213/http://www.agentservice.it/guides.aspx

[13] Imagem arquivada disponível em https://web.archive.org/web/20151002005241/http://www.agentservice.it/forum/

Tabela 3.1: Comparação Esquematizada dos Sistemas Multi-Agente analisados

	JIAC	**JADE**	**SPADE**	**AgentService**
Linguagem de Programação	Java	Java	Python	C#
Comunicação	Apache ActiveMQ	IMTP	XMPP	SOA
Segurança-Autenticidade	Sim	Sim	Sim	Sim (em WCF)
Segurança-Confidencialidade	Sim	Sim	Sim	Sim (em WCF)
Segurança-Integridade	Sim	Sim	Sim	Sim (em WCF)
Suporte FIPA	Sim	Sim	Sim	Sim
Suporte Agentes Externos	Não	Não	Não	Sim
Receção mensagens por eventos	Sim	Sim	Sim	Não
Código Aberto	Sim	Sim	Sim	Sim
Qualidade documentação	Boa	Sem atualizações recentes	Razoável	Boa
Suporte e Comunidade	Pouco ativa mas com resposta rápida	Pouco ativa mas com resposta rápida	Ativa e com resposta rápida	Inativo
Estado de Desenvolvimento	Ativo	Ativo	Ativo	Parado desde 2009
Ultima Versão	Dez. 2017	Jun. 2017	Mai. 2020	Parado desde 2009

industria [Braga et al., 2008]; um sistema que permite o controlo e monitorização de uma rede de sensores sem fio (*wireless*), que permite, por exemplo, recolher dados de condições físicas de um determinado ambiente [Zak et al., 2012]; *Agreement Negotiation in Normative and Trust-enabled Environments* (ANTE), um sistema que permite acordos entre agentes verificando o cumprimento de ambas as partes, de forma a melhorar o processo de negociação futuro [Lopes Cardoso et al., 2016]. Por fim, o SPADE é utilizado para: uma plataforma para a criação de um simulador, SimFleet, para o uso de frotas automóveis de uma forma mais descentralizada e seguindo as necessidades dos dias de hoje [Carrascosa et al., 2019]; um sistema de recomendação que analisa o comportamento das pessoas quando visitam coleções em museus [Gelvez García et al., 2019].

3.2 Plataformas para *Middleware* Orientados a Mensagens

Foi feito um estudo das plataformas de envio de mensagens existentes no mercado, quais as suas caraterísticas, funcionalidades, vantagens e desvantagens. Este estudo baseou-se nos protocolos já estudados em 2.4. O protocolo RestMS, segundo o seu website, apresenta três servidores que implementam o protocolo (Ahkera, Zyre e uma implementação em Perl) [RestMS, 2008]. No entanto, segundo a nossa pesquisa, o Ahkera[14], desenvolvido em Python, parece descontinuado, sem possibilidade de acesso ao GitHub do projeto. O Zyre[15], que já foi uma implementação de RestMS, tem neste momento o website ligado a um projeto que utiliza ZeroMQ para assegurar comunicação de grupos em redes de área local [Zyre Project, 2020]. A implementação disponível em Perl não recebe atualizações desde maio de 2009, pelo que o projeto parece abandonado [Pieter Hintjens, 2009]. Assim, decidimos descartar este protocolo, por considerarmos que não seria uma boa opção para esta dissertação.

[14]Mais informação disponível em http://t-lo.github.io/ahkera/

[15]Imagem arquivada disponível em https://web.archive.org/web/20100225053756/http://www.zyre.com/

O protocolo XMPP, por ter como objetivo original a troca de mensagens entre pessoas, não foi desenhado para apresentar elevado desempenho (baixas latências e elevada taxa de mensagens). Tal como referido em [Gilchrist, 2016], o desempenho do XMPP é baseado na perceção humana de tempo real (segundos), diferentes das necessidades de simulação em tempo real (milissegundos ou mesmo microssegundos). Para além disso, o uso de mensagens em XML tende a tornar as mensagens maiores e necessitar de maior processamento, o que tem impacto no desempenho apresentado [Wang et al., 2013]. Assim, decidimos descartar este protocolo, por considerarmos que também não seria uma boa opção para esta dissertação.

O protocolo MQTT, tal como referido em [Gilchrist, 2016], não foi desenhado para elevado desempenho, mas sim para ser eficiente em termos energéticos, pelo que as latências de comunicação entre cliente e servidor podem atingir os segundos. Para além disso, como referido em [Luzuriaga et al., 2015b], este protocolo apesar de bastante apropriado para ambientes com limitações de largura de banda para a comunicação, deve ser descartado em detrimento de outros, como o AMQP, para sistemas mais complexos de troca de mensagens. Assim, decidimos descartar também este protocolo.

O protocolo DDS apresenta algumas implementações não comerciais que poderiam ser usadas na plataforma do LIACC: OpenDDS, Eclipse Cyclone DDS e Vortex OpenSplice Community Edition. No entanto, o Eclipse Cyclone DDS na versão atual, 0.6.0, não apresenta suporte para comunicação segura (está agendada a introdução desta funcionalidade na próxima versão, 0.7.0[16]) [Eclipse Foundation, 2020]. Para além disso, também não disponibiliza um cliente para C#, o que impede a sua utilização na plataforma do LIACC. Da mesma forma, o OpenDDS, apesar de apresentar capacidade de comunicar de forma segura, de acordo com a especificação[17] do OMG, não disponibiliza um cliente para C# [OpenDDS Project, 2020]. O Vortex OpenSplice Community Edition disponibiliza cliente para C#, bem como a capacidade de comunicar de forma segura [Vortex OpenSplice, 2019]. No entanto, esta implementação apresenta alguns problemas como a falha na descoberta de nós [Damasceno, 2020]. Devido ao uso de *multicast* este mecanismo apenas funciona numa mesma rede, sendo no entanto relatadas situações em que dentro de uma mesma rede o mecanismo não funcionou. O Vortex OpenSplice apresenta uma versão comercial em que estes problemas parecem estar resolvidos, mas que não é uma opção para o uso na plataforma do LIACC [Damasceno, 2020]. Por este motivos, decidimos descartar este protocolo.

O protocolo STOMP, tal como o XMPP, é baseado na troca de mensagens de texto. O uso de protocolos baseados na troca de mensagens de texto torna o processo bastante simples e permite a leitura por humanos. No entanto, tende a tornar as mensagens maiores, bem como a necessitar de maior processamento, o que tem impacto no desempenho quando comparado com outros protocolos que permitem o envio de mensagens em binário (como o AMQP) [Wang et al., 2013].

Para os restantes protocolos apresentados em 2.4 foram estudadas plataformas que os implementam. Procuramos plataformas que sendo mais focadas pela literatura, sejam de uso grátis,

[16]Mais informação disponível em https://projects.eclipse.org/projects/iot.cyclonedds/releases/0.7.0-coquette

[17]Mais informação disponível em https://www.omg.org/spec/DDS-SECURITY/1.1/PDF

apresentem desenvolvimento ativo e suporte para C#. Para o protocolo AMQP foi estudado o RabbitMQ, que apesar de ser poliglota em protocolos de comunicação, foi desenvolvido de raiz com o protocolo AMQP. Para além disso, é um dos *middleware* orientados a mensagens mais referenciados pela literatura. O ActiveMQ, que também é um dos *middleware* mais referenciados, usa de raiz o protocolo JMS 2.0 com pequenas modificações, pelo que foi utilizado com esse protocolo. Foram também estudados o Apache Kafka e o ZeroMQ, que apresentam protocolos próprios mas bastantes usados e referenciados.

3.2.1 ZeroMQ

O **ZeroMQ** é uma biblioteca de comunicação de código aberto desenvolvida pela iMatix Corporation sob uma licença LGPLv3 e com implementações para diversas linguagens como C, C++, C#, Java e Python [Marcos, 2016, Lauener and Sliwinski, 2017].

Esta biblioteca utiliza comunicação orientada a mensagens, em contraste com protocolos padrão, baseados em *stream* (TCP) ou em datagrama (UDP) [Grigorik, 2010]. Assim, o *buffering* e a divisão das mensagens durante o transporte, associado ao envio de uma mensagem, é responsabilidade da biblioteca. Suporta comunicação dentro do mesmo processo (*in-process*), entre processos (*inter-process*), TCP e *multicast*. A Figura 3.1 mostra o processo de envio de uma mensagem entre dois clientes.

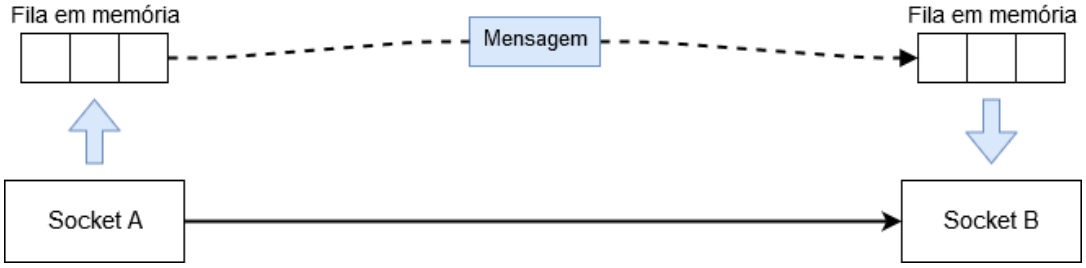


Figura 3.1: ZeroMQ *Sockets* (Adaptado de [Vojkovic, 2014])

A política do ZeroMQ é minimizar todos os recursos usados no processo de comunicação, sendo o seu foco principal diminuir a complexidade dos mecanismos em vez de acrescentar novas funcionalidades [ZeroMQ Team, 2019b]. Assim, a biblioteca apresenta uma arquitetura sem *broker* (*brokerless*), utilizando ligações diretamente entre emissor e recetor, num estilo P2P (*Peer-to-Peer*) [Marcos, 2016]. Devido à não existência de um *broker*, ou seja, um ponto intermédio entre produtor e consumidor, esta arquitetura tem, em teoria, duas vantagens: permite um maior número de mensagens por segundo, porque, nestes sistemas, o ponto de *bottleneck* para um número elevado de mensagens é, na maioria dos casos, a falta de capacidade do *broker*; também a latência poderá ser melhorada porque, numa arquitetura com *broker*, a mensagem tem de ser enviada para o *broker*, ser processada e enviada só depois para o destinatário, enquanto que nesta arquitetura apenas existe a latência de envio entre origem e destino.

Por outro lado, com o uso de uma arquitetura sem *broker* surge um problema, associado aos cenários de *broadcast*, a que a documentação do ZeroMQ chamou "*Dynamic Discovery Problem*"[ZeroMQ Team, 2019a]. Este problema está relacionado com a dificuldade da descoberta de nós num sistema que se pretende distribuído e em que os nós são dinâmicos (vão entrando e saindo). Num cenário de *broadcast* em que temos apenas um produtor, cada consumidor interessado apenas precisa de conhecer esse produtor. No entanto, se surgirem mais produtores, como podemos informar os consumidores desses novos produtores (para que se possam ligar)? A documentação do ZeroMQ define como solução a introdução de um ponto intermédio (*proxy*/*broker*) a que todos os consumidores e produtores se ligam, resolvendo o problema. Assim, no cenário de *broadcast* o ZeroMQ tem um funcionamento semelhante ao de um *middleware* com uma arquitetura que usa *broker*.

Um consumidor, depois de abrir uma ligação para o produtor, pode subscrever mensagens vindas deste, filtrando as mensagens pelo seu conteúdo inicial (prefixo), através do uso de expressões regulares [Hintjens, 2013]. Assim, o ZeroMQ permite a subscrição usando o modelo comum baseado em tópico e o modelo baseado em conteúdo através do uso de expressões regulares.

Esta biblioteca implementa desde a versão 4.0 um mecanismo de segurança chamado CurveZMQ, garantindo autenticidade, confidencialidade e integridade na troca de mensagens. Como tal, consegue prevenir alguns dos ataques mais comuns, como: alteração de dados, ataques de repetição, ataques de amplificação, ataques do tipo intermediário, ataques de roubo de chave, ataques de identidade e certos ataques de *Denial Of Service* [Curve Project, 2019]. A implementação é baseada em criptografia de curva elíptica, o que permite o mesmo nível de segurança com chaves bastante mais pequenas quando comparado com outros algoritmos de chave pública (secção 2.5), e consequentemente melhor desempenho.

A documentação[18] que encontramos na página web é bastante detalhada na descrição do funcionamento do *middleware*, bem como da sua arquitetura. A sua comunidade de suporte[19], permite colocar questões usando uma lista de discussão ou através de um canal IRC (*Internet Relay Chat*). A lista de discussão teve, em Maio de 2020, 29 interações, com as questões normalmente respondidas de forma rápida.

3.2.2 RabbitMQ

O **RabbitMQ** é uma plataforma de comunicação de código aberto desenvolvida pela Pivotal Software sob a licença Mozilla Public License [Pivotal RabbitMQ, 2019a]. Da plataforma fazem parte um servidor, disponibilizado para diversos sistemas operativos (Windows, MacOS e Linux), bem como clientes disponíveis em várias linguagens de programação como C, C++, C#, Java e Python [Marcos, 2016]. A plataforma tem suporte para os diversos modelos de partilha como *Publish-Subscribe* e *Point-to-Point* [Marcos, 2016].

O RabbitMQ permite o uso de vários protocolos orientados a mensagens como o AMQP 0.9.1 e 1.0, o STOMP 1.0 a 1.2 e o MQTT 3.1.1 [Pivotal RabbitMQ, 2020e]. No entanto, o RabbitMQ

[18] Mais informação disponível em `http://zguide.zeromq.org/page:all`
[19] Mais informação disponível em `http://wiki.zeromq.org/docs:mailing-lists`

foi originalmente desenvolvido usando o AMQP versão 0.9.1 e é esse o protocolo principal que implementa. Os restante são suportados via *plugins*. Assim, e usando o AMQP 0.9.1, no RabbitMQ o envio de mensagens de um produtor para um consumidor tem sempre de passar por um ponto intermédio, a que se dá o nome de *Exchange*. Podemos pensar no exchange como um posto de correios, assim uma mensagem é enviada para o *exchange* e este trata do envio da mensagem para as filas de destino. A seleção das filas de destino é feita em função do tipo de *exchange*. Os tipos de *exchanges* que existem são *Direct*, *Fanout*, *Topic* e *Headers*, descritos na secção 2.4. A plataforma permite a subscrição usando o modelo baseado em tópico e o modelo baseado em conteúdo (usando expressões regulares no *exchange Topic*).

O RabbitMQ apresenta de raiz um distribuidor de carga (*load balancer*) que distribui as mensagens pelos diversos consumidores ligados a uma fila de mensagens, permitindo uma distribuição uniforme da carga, isto é, evitando sobrecarga dos consumidores [Marcos, 2016].

O RabbitMQ possui ainda uma interface gráfica de gestão para controlo e monitorização dos diferentes aspetos do *exchange*. Esta interface está disponível através de um servidor HTTP na máquina onde se encontra o servidor da plataforma [Marcos, 2016, Pivotal RabbitMQ, 2019b]

Em relação a segurança, esta plataforma suporta de raiz autenticação e autorização no acesso aos dados, bem como o uso de SSL para estabelecer um canal encriptado entre emissor e recetor. O uso de SSL na comunicação permite assegurar confidencialidade, integridade e autenticidade no transporte das mensagens [Freier et al., 2011].

A documentação[20] que encontramos na sua página web é bastante detalhada na descrição do funcionamento do *middleware*, da arquitetura e protocolos suportados. Para além disso, são fornecidos tutoriais[21] de utilização para diferentes cenários típicos de comunicação e para um vasto conjunto de linguagens de programação, um fator diferenciador em relação aos outros *middleware*. A sua comunidade de suporte[22], permite colocar questões através de: uma lista de discussão, do GitHub e através de um canal IRC ou Slack. A lista de discussão é bastante ativa, com mais de 130 questões no mês de Maio de 2020, com as questões normalmente respondidas de forma rápida.

3.2.3 ActiveMQ

O **ActiveMQ** é uma biblioteca de comunicação de código aberto desenvolvida pela Apache Software Foundation sob a licença Apache License 2.0 [Apache Software Foundation, 2019g]. Esta biblioteca pode ser usada nos diversos sistemas operativos (Windows, MacOS e Linux) [Apache Software Foundation, 2019j]. Existem, neste momento, duas versões desta biblioteca, a versão ActiveMQ 5 "Classic"e a versão ActiveMQ Artemis. A versão Artemis será no futuro a sucessora do ActiveMQ 5 "Classic"como ActiveMQ 6 [Apache Software Foundation, 2019e]. Assim, será analisada em mais detalhe a versão que se espera que fique mais tempo no ativo (versão Artemis).

A plataforma tem suporte para os diversos modelos de partilha como *Publish-Subscribe* e *Point-to-Point* [Apache Software Foundation, 2019b]. Para além disso, permite a subscrição

[20] Mais informação disponível em https://www.rabbitmq.com/documentation.html
[21] Mais informação disponível em https://www.rabbitmq.com/getstarted.html
[22] Mais informação disponível em https://www.rabbitmq.com/contact.html#community-resources

usando o modelo comum baseado em tópico e o modelo baseado em conteúdo através do uso de expressões regulares.

No ActiveMQ Artemis é utilizado o conceito de *address* (endereço) para fazer a associação a uma fila ou a um conjunto de filas. As mensagens têm um endereço que indica para onde devem ser enviadas. O envio de uma mensagem para um determinado endereço pode corresponder ao envio, por parte do servidor/*broker*, da mensagem para uma ou mais filas que estão associadas a esse endereço. Existem dois tipos de *routing* de uma mensagem: *multicast*, em que cada mensagem é enviada para todas as filas que estão associados àquele endereço; *anycast*[23], em que a mensagem é enviada apenas para uma fila. Quando várias filas estão associadas ao endereço, então o envio vai funcionar de uma forma *round-robin*, isto é, alternadamente e de forma balanceada vai colocando novas mensagens em diferentes filas [Apache Software Foundation, 2019d].

No ActiveMQ Artemis existem algumas propriedades que as mensagens podem apresentar. Assim, uma mensagem pode:

- ser persistente ("*durable*") ou não persistente ("*non durable*"), indicando se esta deve ou não sobreviver a um *crash* ou *restart* do servidor, respetivamente [Apache Software Foundation, 2019d];
- uma mensagem pode apresentar prioridade associada, especificada por um valor entre 0 e 9, em que 0 representa o nível mais baixo de prioridade e 9 o nível mais alto [Apache Software Foundation, 2019d];
- apresentar um tempo de expiração, a partir do qual a mensagem não é mais entregue pelo *broker* [Apache Software Foundation, 2019d];
- conter um *timestamp* associado, indicando quando foi enviada [Apache Software Foundation, 2019d];
- apresentar a possibilidade de aplicação de filtros que são escritos utilizando uma sintaxe de expressões regulares específica, *Apache Artemis Filter Expressions*[24]. Estes filtros podem ser aplicados antes da entrada numa fila (*Queue Filter*) ou só depois da mensagem ter chegado a uma fila (*Consumer Filters*) [Apache Software Foundation, 2019b].

Ao nível de segurança esta biblioteca apresenta os seguintes mecanismos [Apache Software Foundation, 2019k]:

- **Autorização** em que é possível definir que grupo de utilizadores pode realizar certas ações críticas. Assim, é possível controlar quais os utilizadores que podem, por exemplo, ler ou escrever numa fila. Este modelo de permissões pode ser aplicado diretamente a um endereço, ou a um conjunto de endereços utilizando *wildcard match*[25].

[23] Mais informação disponível em https://www.cloudflare.com/learning/cdn/glossary/anycast-network/

[24] Mais informação disponível em https://activemq.apache.org/components/artemis/documentation/latest/filter-expressions.html

[25] Mais informação disponível em https://activemq.apache.org/components/artemis/documentation/latest/wildcard-syntax.html

- **Confidencialidade e Integridade** através do uso de SSL, que encripta e assina os conteúdo das mensagens enviadas.
- **Autenticidade** usando o mecanismo tradicional (*username* e *password*), ou utilizando o certificado usado para o SSL. Existe também um modo de dupla autenticação em que autenticação é possível usando o certificado SSL ou, quando não estão a ser usadas ligações SSL, usando o mecanismo tradicional.

Todo o funcionamento descrito diz respeito ao funcionamento *Core* da biblioteca ActiveMQ Artemis, sendo este o que fornece mais funcionalidades e de uma forma mais simplificada [Apache Software Foundation, 2019d]. Este funcionamento segue o protocolo Java Messaging Service (JMS) 2.0, com algumas adaptações. A biblioteca suporta, no entanto, outros protocolos padrão como: JMS 1.1 e 2.0 padrão[26], AMQP 1.0[27], MQTT 3.1 e 3.1.1[28] e STOMP 1.0, 1.1 e 1.2[29]. Para além disso, permite a um cliente da versão ActiveMQ 5 "Classic"comunicar com um servidor da versão Artemis através do protocolo OpenWire [Apache Software Foundation, 2019h].

A documentação[30] que encontramos na sua página web é bastante detalhada na descrição do funcionamento do *middleware*, da arquitetura e protocolos suportados. A sua comunidade de suporte[31], permite colocar questões através de: listas de discussão e através de um canal IRC ou Slack. A lista de discussão para questões gerais é bastante ativa, com quase 100 interações no mês de Maio de 2020, com as questões normalmente respondidas de forma rápida.

3.2.4 Apache Kafka

O **Apache Kafka** é uma plataforma de comunicação de código aberto desenvolvida pelo Linkedin para a Apache Software Foundation que se encontra sob uma licença Apache License 2.0. [Marcos, 2016]. Esta plataforma pode ser usada nos diversos sistemas operativos [Apache Software Foundation, 2019i], tendo clientes desenvolvidos para as diversas linguagens de programação (C, C++, .NET, Java e Python) [Rao, 2019].

O Apache Kafka corre como um *cluster*, podendo estar distribuído por vários servidores de forma a assegurar mais tolerância a falhas. O *cluster* guarda *streams* de *records* em categorias, os tópicos. Um *record* corresponde a uma chave, um valor e um *timestamp* [Apache Software Foundation, 2019f].

Um tópico funciona como um endereço no qual os *records* são publicados pelos produtores. Um tópico é constituído por várias partições. A Figura 3.2 mostra a arquitetura base da plataforma.

[26] Mais informação disponível em https://activemq.apache.org/components/artemis/documentation/latest/using-jms.html
[27] Mais informação disponível em https://activemq.apache.org/components/artemis/documentation/latest/amqp.html
[28] Mais informação disponível em https://activemq.apache.org/components/artemis/documentation/latest/mqtt.html
[29] Mais informação disponível em https://activemq.apache.org/components/artemis/documentation/latest/stomp.html
[30] Mais informação disponível em https://activemq.apache.org/components/artemis/documentation/
[31] Mais informação disponível em https://activemq.apache.org/contact/

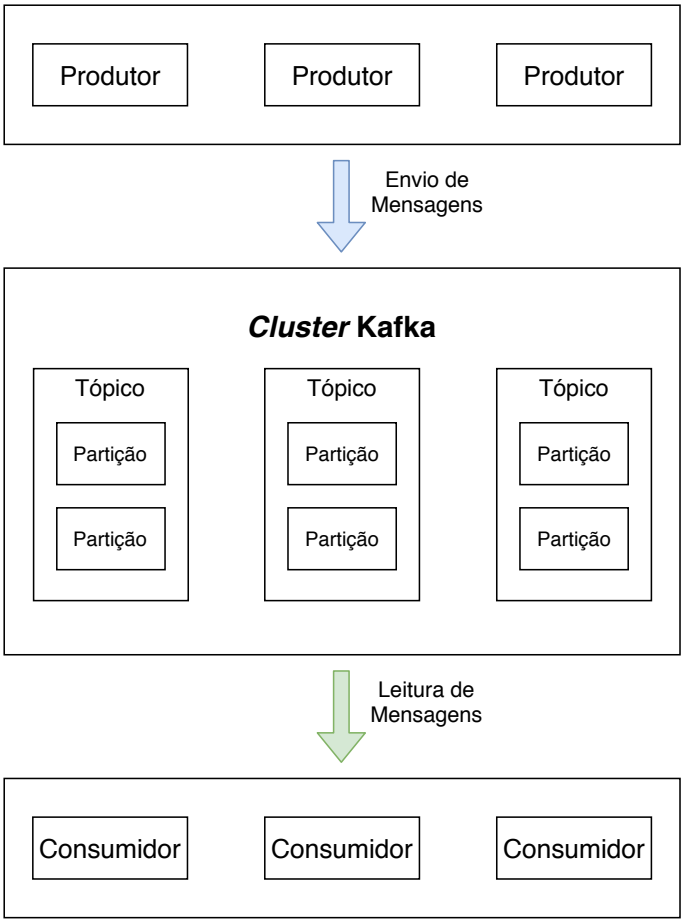


Figura 3.2: Arquitetura Apache Kafka

Esta divisão é feita principalmente por duas razões: permitir a distribuição de um tópico por vários servidores, em que por exemplo, cada servidor tem uma das partições do tópico; e porque as partições funcionam como meio para atingir paralelismo [Apache Software Foundation, 2019f]. Cada partição é constituída por *records* que foram aí publicados. Um produtor é responsável pela escolha da partição, em que o *record* que produziu será publicado. Esta escolha pode ser feita de forma balanceada (*round-robin*) ou através de uma outra qualquer heurística. A partição mantém os *records* que lhe foram colocados de forma ordenada, usando um identificador único e sequencial para cada *record* adicionado. Os *records* publicados são mantidos durante um período de tempo configurável, estando durante esse período disponíveis para consumo, depois são descartados para libertar espaço [Apache Software Foundation, 2019f]. A documentação refere, no entanto, que o desempenho é mantido constante, no que diz respeito ao tamanho dos dados, sendo possível manter os dados durante bastante tempo sem que isso tenha impacto no desempenho da aplicação.

A leitura pelos consumidores pode ser feita durante o intervalo de tempo em que um *record* está disponível, inclusive mais do que uma vez se assim o desejar. Um consumidor, para fazer a leitura, precisa de especificar o identificador do *record* que pretende ler da partição. Num consumo normal, o consumidor irá ler cada um dos *records* iterativamente, incrementando, portanto, o identificador do *record* que pede. No entanto, como o controlo sobre que *record* ler é totalmente

do consumidor, este pode, em determinadas situações, utilizar uma outra ordem não iterativa. [Apache Software Foundation, 2019f] A comunicação entre clientes (produtores e consumidores) e o servidor usa exclusivamente o protocolo TCP.

O Apache Kafka replica cada partição por um número configurável de servidores, para aumentar a resistência a falhas. Para cada partição existe um servidor que é líder, responsável pelas leituras e escritas nessa partição, e um conjunto de outros servidores (seguidores) que apenas replicam o conteúdo do líder. Se o servidor líder falhar, um dos seguidores irá assumir a liderança daquela partição. Num determinado momento, um servidor pode ser líder de algumas partições e seguidor de outras partições. Existe um balanceamento de carga para que cada servidor de um dado *cluster* tenha à sua responsabilidade, como líder ou seguidor, um número de partições semelhante ao dos outros servidores [Apache Software Foundation, 2019f].

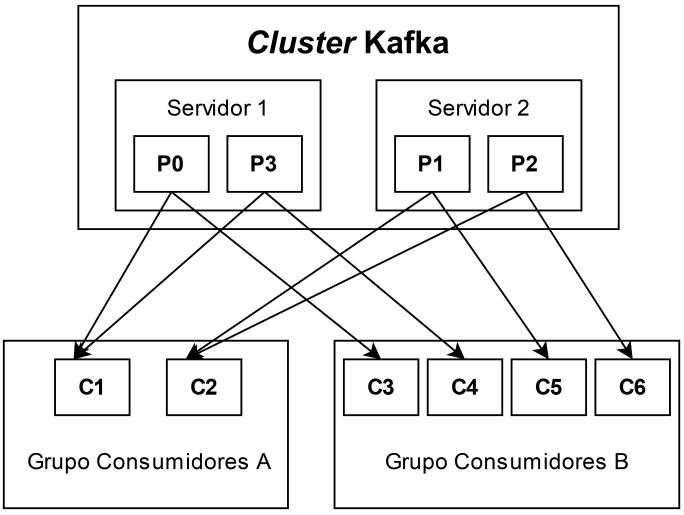


Figura 3.3: Grupos de consumidores em Apache Kafka (Adaptado de [Apache Software Foundation, 2019c])

Em Apache Kafka um consumidor subscreve alterações aos tópicos [Kreps, 2014]. Os consumidores são colocados em grupos, e quando surge um novo *record* num tópico subscrito apenas um dos elementos do grupo irá ser notificado. Na Figura 3.3 é possível ver que cada partição está ligada aos dois grupos de consumidores existentes, para que chegue a ambos, mas apenas a um consumidor de cada. A subscrição pode ser feita usando o modelo comum baseado em tópico, mas também o modelo baseado em conteúdo através do uso de expressões regulares.

De forma a manter uma distribuição balanceada entre os consumidores notificados, o Kafka distribui as partições pelos diversos consumidores associados dentro de um grupo. Na Figura 3.3 é possível ver a divisão uniforme das quatro partições existentes. No grupo de consumidores A como temos apenas dois consumidores, cada um recebe mensagens de duas partições. No grupo de consumidores B como temos quatro consumidores, cada um recebe mensagens de apenas uma partição. Quando há a entrada ou saída de consumidores no grupo é feita uma redistribuição das partições [Apache Software Foundation, 2019f].

O Apache Kafka disponibiliza alguns mecanismos no que diz respeito à segurança [Apache Software Foundation, 2019a]:

- **Autorização** nas operações de leitura e escrita nos tópicos. Existe a possibilidade de utilizar o mecanismo existente ou integrar um serviço de autorização externo.
- **Confidencialidade** na comunicação entre clientes (produtores e consumidores) e *brokers* e entre os diversos *brokers*. A confidencialidade é garantida através do uso de SSL.
- **Autenticação** na comunicação entre clientes e *brokers*. A autenticação é garantida através do uso de SSL ou SASL (Simple Authentication and Security Layer).
- **Integridade** na comunicação entre clientes e *brokers*. Apesar da literatura não o referir diretamente, o uso de SSL permite ao Apache Kafka a garantia de integridade nas mensagens trocadas [Freier et al., 2011].

As noções explicadas anteriormente correspondem ao caso geral de uso do Apache Kafka, *streaming* de dados [Apache Software Foundation, 2019f]. No entanto, quando substituímos o conceito de *record* pelo de mensagem, obtemos uma solução totalmente adaptada à troca de mensagens entre agentes. Quando comparamos esta solução com outras que utilizam um intermediário (como o ActiveMQ ou o RabbitMQ) podemos verificar que: ambas garantem que as mensagens podem ser entregues aos consumidores pela mesma ordem pela qual foram recebidas pelo servidor. No entanto, no Apache Kafka esta garantia apenas existe entre mensagens vindas da mesma partição. Por outro lado, o Apache Kafka garante paralelismo real na leitura de mensagens, quando temos um número de partições igual ou superior ao número de consumidores do grupo, visto que cada consumidor pode estar a ler de uma partição diferente.

A documentação[32] que encontramos na página web é bastante detalhada na descrição do funcionamento do *middleware* bem como da sua arquitetura. A sua comunidade de suporte[33], permite colocar questões usando listas de discussão ou através de um canal IRC. As listas de discussão são todas bastante ativas (a lista de questões gerais sobre o Kafka teve no mês de Maio de 2020 mais de 300 questões), sendo as questões normalmente respondidas de forma rápida.

3.2.5 Resumo Comparativo dos *Middleware*

A Tabela 3.2 mostra um resumo comparativo das principais caraterísticas dos *middleware* de comunicação estudados. De destacar o ZeroMQ por ser o único, dos *middleware* estudados, que apresenta uma arquitetura sem o uso de um intermediário/*broker*. Esta arquitetura coloca o ZeroMQ em vantagem em relação aos outros *middleware*, permitindo concluir que, pelo menos em teoria este *middleware* conseguirá apresentar menor latência e maior taxa de transferência de mensagens. Por outro lado, o ZeroMQ é o único que não apresenta a possibilidade de ter prioridade associada às mensagens, o que pode ser importante para alguns usos específicos. É também o único que não apresenta uma interface de gestão que permita a visualização do estado e gestão

[32] Mais informação disponível em https://kafka.apache.org/documentation/
[33] Mais informação disponível em https://kafka.apache.org/contact

Tabela 3.2: Comparação Esquematizada das Plataformas MOM Analisadas

	ZeroMQ	**RabbitMQ**	**ActiveMQ**	**Apache Kafka**
Uso Grátis	Sim	Sim	Sim	Sim
Poliglota (Linguagens de Programação)	Sim	Sim	Sim	Sim
Multiplos Sistemas Operativos	Sim	Sim	Sim	Sim
Comunicação Ponto-a-Ponto	Sim	Sim	Sim	Sim
Comunicação Publish-Subscribe	Sim	Sim	Sim	Sim
Mensagens com prioridade	Não	Sim	Sim	Não
Subscrição por Tópico de Mensagem	Sim	Sim	Sim	Sim
Subscrição por Expressões Regulares	Sim	Sim	Sim	Sim
Independência de Broker	Sim	Não	Não	Não
Segurança - Autenticidade	Sim	Sim	Sim	Sim
Segurança - Confidencialidade	Sim	Sim	Sim	Sim
Segurança - Integridade	Sim	Sim	Sim	Sim
Interface de Gestão	Não	Sim	Sim	Sim
Qualidade documentação	Boa	Muito Boa	Boa	Boa
Suporte e Comunidade	Bastante ativa / resposta rápida	Bastante ativa / resposta rápida	Ativa / resposta rápida	Bastante ativa / resposta rápida
Estado de Desenvolvimento	Ativo	Ativo	Ativo	Ativo
Ultima Versão	Jun. 2020	Jun. 2020	Mai. 2020	Abr. 2020

em tempo real da plataforma, uma vez que não apresenta *broker*. Alguns estudos realizados, nomeadamente [Estrada and Astudillo, 2015] e [Dobbelaere and Esmaili, 2017], mostram que o desempenho do ZeroMQ é realmente superior quando comparado com o do RabbitMQ e do Apache Kafka. Espera-se também, segundo esses estudos, que o RabbitMQ apresente melhor desempenho quando comparado com o Apache Kafka.

Estes *middleware* apresentam aplicabilidade nas mais diversas aplicações no mundo real, bem como em sistemas relacionado com MAS. O ZeroMQ é utilizado para: uma arquitetura de transmissão de dados para sistemas com múltiplos UAV (veículo aéreo não tripulado) [Uk et al., 2018]; e num sistema para automatizar decisões ao nível da missão para UxAS (Unmanned Systems Autonomy Services) [Li et al., 2018].

O RabbitMQ foi utilizado como *middleware* para: um sistema de organização temporal de tarefas baseado na informação das tarefas atuais e no seu desempenho [Khegai et al., 2015]; e num sistema que permite testar sistema multi-agente usando um arquitetura *publish-subscribe* [Nascimento et al., 2017].

O ActiveMQ foi utilizado como *middleware* para vários projetos, nomeadamente: uma *framework* chamada Decentralized Coordination Framework for Various Multi-Agent Systems (DeCoF) que permite a adaptação dos agentes às alterações do ambiente, utilizando coordenação distribuída em MAS [Preisler et al., 2016]; e no JIAC que analisámos anteriormente.

O Apache Kafka foi também usado em projetos relacionados com MAS como: uma *framework* para análise de dados na Industria 4.0 [Peres et al., 2016]; e uma aplicação MAS distribuída para o processamento continuo de grandes quantidade de dados [Shashaj et al., 2019].

Capítulo 4

Arquitetura e Proposta de Solução

Este capítulo apresenta a arquitetura básica da solução que propomos. Está dividido em quatro grupos principais - descrição do problema, *middleware* vs MAS, testes de desempenho aos *middleware* de comunicação e MAS, e proposta de solução. Começamos por reforçar o problema que tentamos resolver na plataforma do LIACC e noutras plataformas do mesmo género, mostrando depois duas possíveis soluções a seguir. Como ajuda à decisão foram realizados alguns testes de desempenho, dando origem à solução proposta.

4.1 Descrição do Problema

A plataforma do LIACC procura um *middleware* que lhe permita melhorar o desempenho na troca de mensagens e comunicar de forma segura. Este *middleware* deve permitir manter as normas FIPA para a comunicação e manter o suporte para agentes externos. Em geral, com o crescente uso dos sistemas multi-agente nas mais diversas aplicações e áreas, nomeadamente na Industria 4.0 [Pires et al., 2018, Ghadimi et al., 2019] e na gestão de sistemas de energia [Xie and Liu, 2017], surge a necessidade de comunicação mais eficiente e segura em MAS.

Num sistema com centenas ou milhares de agentes em comunicação constante e simultânea é necessário um *middleware* que satisfaça as necessidades de baixa latência e elevada taxa de transferência de mensagens [Leitão et al., 2013]. Esta necessidade intensifica-se ainda mais em plataformas de simulação, como a plataforma do LIACC, em que se tenta assegurar comunicação quase em tempo real [Calvaresi et al., 2017].

Para além disso, deve ser assegurada a troca mensagens de forma segura entre agentes que estão possivelmente em pontos geográficos diferentes e que precisam de trocar mensagens confidenciais. O mesmo acontece em ambientes competitivos em que agentes maliciosos podem tentar alterar o conteúdo das mensagens, ou enviar mensagens em nome de outro agente. Num sistema em que sejam usadas equipas de agentes, como na plataforma do LIACC, isto torna-se ainda mais perigoso porque, apenas a adulteração da informação perante um agente, pode propagar essa informação falsa pelos restantes membros da equipa [Feng et al., 2016]. Na literatura encontramos

diversas referências às necessidades crescentes do uso de segurança em MAS, sendo a comunicação segura um dos elementos mais referidos e importantes, nomeadamente: sistemas MAS usados para o comércio eletrónico, que tem crescido nos últimos anos, e as vulnerabilidades existentes neste tipo de aplicações [Briones et al., 2016]; sistemas MAS na área da saúde e a necessidade de trocar os dados dos paciente de forma segura, como num sistema que permite o diagnóstico e a classificação de diabetes de forma remota [Tangod and Kulkarni, 2018]. Para além disso tem sido crescente o aumento da perceção que a segurança deve ser um fator presente no desenvolvimento de sistemas multi-agente desde o seu início, e não como uma melhoria ou adição secundária [Hedin and Moradian, 2015, Subburaj and Urban, 2018].

A possibilidade de uso de agentes externos numa plataforma multi-agente é algo que consideramos bastante importante para o desenvolvimento de aplicações usando as funcionalidades de um sistema multi-agente. Este tipo de suporte permite introduzir as vantagens e funcionalidades de um sistema multi-agente em aplicações legadas [Oprea, 2004]. Destas funcionalidades destacam-se a distribuição, a localização e comunicação usando um padrão bastante conhecido, o padrão definido pela FIPA para sistemas multi-agente. Para além do seu uso na plataforma do LIACC, esta era uma das principais funcionalidades diferenciadoras do AgentService [The AgentService Team, 2008]. Na nossa pesquisa não foram encontradas outras plataformas multi-agente com suporte explicito para o fazer, mas pensamos que esta é uma funcionalidade importante para a transformação de aplicações legadas, sendo um bom contributo para a comunidade. As plataformas multi-agente permitem a distribuição dos seus agentes por diversas máquinas. Isto pode ser feito dentro de uma mesma plataforma, como no caso do JADE em que a distribuição é feita usando *containers* em máquinas diferentes. Ou então, usando o mecanismo de federação, com várias instâncias da plataforma a correrem em máquinas diferentes. Com o uso de agentes externos surge a necessidade de distribuir também as aplicações das quais os agentes fazem parte. Nesse sentido, a solução a desenvolver deve, mantendo a típica funcionalidade dos MAS, permitir distribuir os agentes e as aplicações das quais estes fazem parte.

De forma a solucionar os problemas referidos existem duas possíveis soluções: a substituição do AgentService por uma nova plataforma multi-agente; ou a substituição apenas do *middleware* de comunicação do AgentService por um *middleware* de comunicação orientado a mensagens.

4.2 *Middleware* vs MAS

O estudo das plataformas multi-agente na secção 3.1 permitiu perceber que estas apresentam alguns problemas no que diz respeito aos *middleware* de comunicação utilizados, possivelmente por terem sido desenhadas e desenvolvidas há já alguns anos. O JADE por exemplo, utiliza um *middleware* de comunicação proprietário que dificulta uma possível interoperabilidade com outras plataformas. O AgentService apresenta o problema conhecido da falta de um mecanismo eficiente para a receção das mensagens. Com as atualizações que foram sofrendo, o SPADE e o JIAC apresentam *middleware* mais recentes e orientados a mensagens (XMPP e Apache ActiveMQ, respetivamente), pelo que seriam os mais apropriados para a nossa solução.

A substituição do AgentService por uma destas plataformas multi-agente permite que a plataforma do LIACC passe a ter integrada uma plataforma MAS que tem desenvolvimento ativo e suporte pela comunidade. Isto permite que no futuro possam existir atualizações que melhorem o desempenho apresentado por esses MAS bem como a introdução de novas funcionalidades, das quais a plataforma do LIACC pode beneficiar. Por outro lado, nenhuma das plataformas MAS estudadas apresenta suporte para C# pelo que a sua integração implica o desenvolvimento de um *wrapper* que o permita. O JADE apresenta um *plugin* que lhe daria esse suporte, mas como já descrito o JADE-LEAP para C# apresenta bastantes problemas e está obsoleto. O desenvolvimento deste *wrapper*, pelo menos para todas as funcionalidades usadas pela plataforma do LIACC e com baixo impacto no desempenho, torna a solução bastante mais complexa e arriscada. Para além disso, nenhuma das plataformas MAS estudadas apresenta suporte para agentes externos, como o AgentService, pelo que o *wrapper* desenvolvido teria de também introduzir suporte para esta funcionalidade. No caso do JADE e do SPADE seriam ainda necessárias algumas adaptações para a comunicação segura com o DF e o AMS.

Os *middleware* orientados a mensagens estudados na secção 3.2 apresentam caraterísticas muito semelhantes e que vão de encontro ao que era procurado, nomeadamente a segurança embebida e o desempenho teórico. Será necessária uma avaliação do seu desempenho, no contexto da plataforma do LIACC, para uma escolha mais fundamentada de alguma delas. De referir apenas a vantagem que o ZeroMQ apresenta, devido a ser o único *middleware* com arquitetura sem broker, permitindo obter menor latência (mensagem vai diretamente para o seu recetor ou recetores) e maior número de mensagens enviadas por segundo (os brokers são, normalmente o ponto de bottleneck do sistema por estarem sujeitos a bastante carga).

A substituição apenas do *middleware* de comunicação do AgentService, por um deste *middleware*, permitiria que em teoria o desempenho da comunicação fosse superior, por se tratarem de tecnologias mais recentes e específicas para a comunicação com elevado desempenho. Para além disso, o uso de um destes *middleware*, quando comparado com o JADE e outros MAS que apresentam protocolos de comunicação proprietários, permite uma maior interoperabilidade com outras plataformas MAS e outras aplicações. Por outro lado, o uso de duas plataformas (uma para comunicação e outra para as restantes funcionalidades) implica maior manutenção, por exemplo, para manter as 2 plataformas constantemente atualizadas. Para além disso, seria necessário introduzir um *wrapper* que permite a comunicação segura com o DF e o AMS do AgentService.

De forma a ajudar na decisão de qual a solução a seguir, e que plataforma escolher, é necessário analisar o desempenho real que cada plataforma apresenta. Assim, foram realizados testes de desempenho aos *middleware* e às plataformas MAS estudadas.

4.3 Testes de Desempenho aos *Middleware* de Comunicação e MAS

Os testes aos *middleware* de comunicação e MAS foram realizados inicialmente sem o uso de segurança na troca de mensagens. Depois, os mesmos testes foram realizados com o uso de segurança, garantindo integridade, confidencialidade e autenticidade.

Os testes foram realizados para três cenários típicos de comunicação entre agentes numa qualquer plataforma: comunicação um-para-um, comunicação um-para-muitos (*broadcast*) e comunicação muitos-para-um. Na comunicação um-para-um foram realizados testes com 10, 50 e 100 pares de agentes a comunicar em simultâneo. O objetivo era perceber o desempenho obtido com o uso de poucos agentes (simulações mais pequenas) e como escalava o desempenho para um maior número de agentes (simulações maiores). No contexto da plataforma do LIACC este tipo de comunicação é usado na maioria das comunicações entre agentes.

Na comunicação um-para-muitos foram realizados testes com: 10 agentes produtores para 10, 50 e 100 agentes consumidores; e 50 agentes produtores para 10 e 50 agentes consumidores. O objetivo era perceber o efeito que o aumento do número de produtores e consumidores tinha no desempenho, de forma a analisar a escalabilidade. No contexto da plataforma do LIACC, este tipo de comunicação surge devido à necessidade de realizar simulações com vários ATC's, que utilizam *broadcast* para comunicar com um conjunto de agentes veículo (por exemplo na comunicação de um acidente ou para a comunicação de outras informações mais frequentes).

Na comunicação muitos-para-um foram realizados testes com 10, 50 e 100 agentes produtores a enviarem mensagens para apenas 1 agente consumidor. O objetivo era perceber como se comportava um agente recetor com o envio de um elevado número de mensagens vindas de um número de agentes que foi sendo aumentado. No contexto da plataforma do LIACC, esta comunicação surge, por exemplo, quando ao receberem a informação de indicação de acidente, por parte de um controlador de tráfego, todos os agentes veículo lhe respondem. Num contexto mais geral de MAS, este tipo de comunicação acontece quando os múltiplos agentes realizam pedidos às páginas amarelas ou páginas brancas em simultâneo.

As mensagens enviadas correspondem ao *timestamp* em milissegundos de envio dessa mensagem, com um tamanho variável entre 6 e 10 bytes. Nas plataformas multi-agente o tamanho das mensagens enviadas é ligeiramente superior, devido a estas necessitarem de seguir o formato FIPA ACL[1] (este tamanho vai ainda depender da ferramenta de serialização usada). Para medir a latência de envio das mensagens é utilizada a diferença entre o *timestamp* de receção e o *timestamp* de envio (que é a própria mensagem). Para o cálculo da taxa de transferência de mensagens foi calculado o tempo de envio do total de mensagens enviadas, através da diferença entre o *timestamp* de envio da última mensagem e o *timestamp* de envio da primeira mensagem. Foram enviadas por cada produtor um grupo com 500 mensagens, de forma iterativa (uma a seguir à outra) com um intervalo de um milissegundo entre cada envio. O uso deste intervalo entre o envio de mensagens, para além de corresponder a um caso mais próximo do real, tem um grande impacto nos valores de desempenho apresentados pelas plataformas.

Os gráficos apresentam, em alguns casos, escalas logarítmicas para os valores de latência e de mensagens/s, devido às grandes diferenças de magnitude dos resultados apresentados pelas diferentes plataformas. Os valores apresentados nas secções que se seguem correspondem à média da realização de 3 medições para cada um dos testes, de forma a reduzir erros nos valores observados.

[1] Mais informação disponível em http://www.fipa.org/specs/fipa00061/SC00061G.html#_Toc26669700

Foram usadas as versões mais recentes dos *brokers*, clientes e plataformas disponíveis à data de realização dos testes: os *brokers* ActiveMQ Artemis[2] 2.10.1, RabbitMQ[3] 3.8.2 e Apache Kafka[4] 2.4.0; os clientes C# Apache.NMS.ActiveMQ[5] 1.7.2, RabbitMQ.Client[6] 5.1.2, Confluent.Kafka[7] 1.3.0, NetMQ[8] 3.3.3.4 e clrzmq4[9] 4.1.0.31; e as plataformas MAS JIAC[10] 5.2.4, JADE[11] 4.5.0 em conjunto com o JADE-S[12] 3.10 e SPADE[13] 3.1.4. Todos os testes foram realizados num CPU Intel 4720HQ@2.6Ghz, 16GB de RAM @ 1600MHz, 500GB SSD, Windows 10 Home.

4.3.1 Testes Sem Segurança

Os resultados dos testes de desempenho sem o uso de segurança são mostrados aqui. Esta primeira abordagem tem como principal objetivo estabelecer um valor de desempenho base para a troca de mensagens nos cenários descritos.

4.3.1.1 Um-para-Um

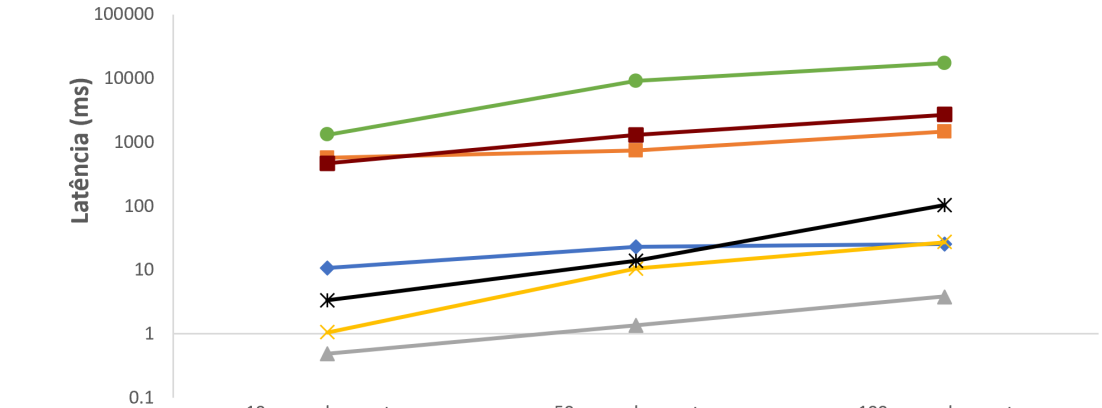
Neste que é um dos cenários mais típicos de comunicação entre agentes, os resultados dos testes de desempenho realizados mostram o ZeroMQ como aquele que apresentou o melhor desempenho em todos os cenários. A sua vantagem teórica, devido a ser o único que apresenta uma arquitetura sem *broker* (*brokerless*), permite que consiga apresentar não só uma menor latência, com cada mensagem a ser enviada diretamente do produtor para o agente consumidor (como pode ser visto na Fig. 4.1), como também maior taxa de transferência de mensagens (como pode ser visto na Fig. 4.2) porque neste tipo de sistemas o ponto de *bottleneck* é, normalmente, o *broker*.

Os valores que o RabbitMQ e o ActiveMQ apresentam são, de forma geral, também bastante bons. A capacidade de envio de mensagens por unidade de tempo é, no entanto, bastante inferior no ActiveMQ, o que de resto é uma constante nos restantes testes.

De destacar os valores bastante elevados de latência que o Apache Kafka apresenta. Estes valores são uma constante também nos restantes cenários de comunicação. Foram utilizados dois clientes C# (Confluent.Kafka 1.3.0 e kafka-net[14] 9.0.65) na tentativa de encontrar melhores resultados, sendo aqui apresentados os resultados do melhor cliente (Confluent.Kafka).

No caso do JADE depois de alguns testes iniciais, usando apenas um processo que lançava todos os agentes (depois cada um corre no seu *thread*) observamos que os valores de latência eram demasiado baixos (por exemplo, 0.082ms para o envio de 100 mensagens consecutivamente entre

[2]Mais informação disponível em https://activemq.apache.org/components/artemis/
[3]Mais informação disponível em https://github.com/rabbitmq/rabbitmq-server
[4]Mais informação disponível em https://kafka.apache.org/downloads
[5]Mais informação disponível em https://www.nuget.org/packages/Apache.NMS.ActiveMQ/
[6]Mais informação disponível em https://github.com/rabbitmq/rabbitmq-dotnet-client
[7]Mais informação disponível em https://github.com/confluentinc/confluent-kafka-dotnet
[8]Mais informação disponível em https://github.com/zeromq/netmq
[9]Mais informação disponível em https://github.com/zeromq/clrzmq4
[10]Mais informação disponível em http://www.jiac.de/agent-frameworks/jiac-v/
[11]Mais informação disponível em https://jade.tilab.com/download/jade/
[12]Mais informação disponível em https://jade.tilab.com/download/add-ons/
[13]Mais informação disponível em https://pypi.org/project/spade/
[14]Mais informação disponível em https://www.nuget.org/packages/kafka-net/



	10 pares de agentes	50 pares de agentes	100 pares de agentes
ActiveMQ	10.71	23.31	25.52
Apache Kafka	573.76	747.28	1475.05
ZeroMQ	0.49	1.35	3.85
RabbitMQ	1.05	10.55	27.30
JADE	3.36	14.12	104.32
JIAC	1317.25	9223.40	17436.54
SPADE	468.16	1301.81	2706.34

Figura 4.1: Latência na Troca de Mensagens - Um-para-Um - Sem Segurança

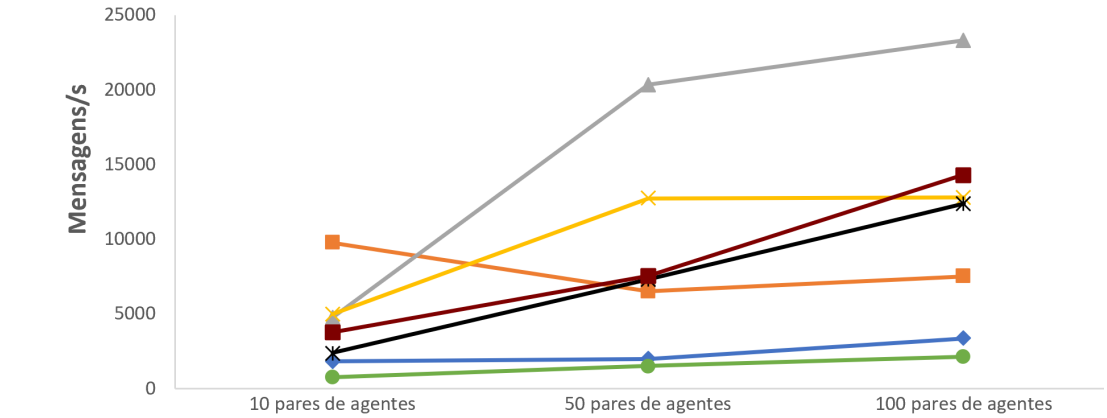
cada par de agentes utilizando 10 pares de agentes). Estes valores demasiado baixos justificam-se talvez através do uso de algum mecanismo intra-processo que otimiza o envio de mensagens nesta situação. Assim e para simular um cenário mais realista e coerente com as restantes plataformas optamos por utilizar dois processos diferentes, um responsável pelo agentes emissores e outro pelos agentes recetores.

De qualquer forma os resultados mostram o JADE como a plataforma MAS que apresenta melhores resultados em termos de latência (ainda que longe dos melhores *middleware*). As restantes plataformas apresentam valores de latência bastante mais elevados, o que de resto é uma constante também noutros cenários de comunicação.

Em termos de escalabilidade na latência, notamos que o ZeroMQ apresenta uma ótima escalabilidade, seguido do ActiveMQ e do RabbitMQ. O JADE tem uma aumento bastante brusco na passagem dos 50 para os 100 pares de agentes (um aumento superior a 7 vezes), pelo que é indicador que poderia não escalar para um maior número de agentes. As restante plataformas, devido aos valores muito elevados de latência que apresentam, não permitem uma análise concreta.

Em termos da taxa de transferência de mensagens, destacar os valores bastante mais baixos do ActiveMQ e do JIAC quando comparados com as outras plataformas. Destacar também os valores do Apache Kafka que descem para os cenários com maior número de agentes, em contraste com as restantes plataformas, apesar de apresentar valores razoáveis.

Em termos de escalabilidade na taxa de transferência de mensagens, o RabbitMQ parece ter atingido um máximo, próximo das 13 mil mensagens enviadas por segundo, para este cenário. O Apache Kafka parece também não conseguir valores acima das 10 mil mensagens enviadas por segundo, diminuindo até nos cenários de maior carga.



	10 pares de agentes	50 pares de agentes	100 pares de agentes
ActiveMQ	1829.60	2004.17	3357.51
Apache Kafka	9762.85	6512.11	7524.60
ZeroMQ	4770.99	20341.74	23320.90
RabbitMQ	4980.08	12742.10	12791.00
JADE	2393.49	7318.50	12376.24
JIAC	762.85	1512.11	2124.60
SPADE	3770.99	7541.74	14320.90

Figura 4.2: Taxa de Transferência de Mensagens - Um-para-Um - Sem Segurança

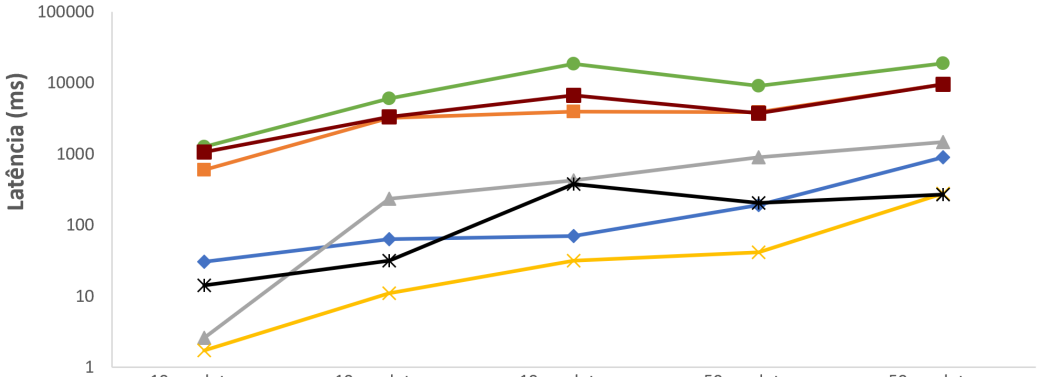
4.3.1.2 Um-para-Muitos

Os resultados dos testes de desempenho realizados mostram que, neste tipo de comunicação, é o RabbitMQ que apresenta menor latência em quase todos os cenários, como pode ser visto na Fig. 4.3. A análise do gráfico permite também notar um grande decréscimo de desempenho no que diz respeito à latência por parte do ZeroMQ. O ZeroMQ que tinha sido superior na comunicação um-para-um, está agora atrás do RabbitMQ, do ActiveMQ e do JADE. Este decréscimo de desempenho por parte do ZeroMQ pode ser explicado com a necessidade do uso de um *broker* para este tipo de comunicação, e consequente perda da vantagem teórica que apresentava.

Neste tipo de comunicação, que coloca mais carga sobre os *middleware* devido ao maior número de mensagens a circular, no ZeroMQ, para evitar uma elevada perda de mensagens, foi necessária a alteração de algumas configurações padrão. Foram aumentados os valores de *ReceiveHighWatermark* e de *SendHighWatermark* que representam o limite de mensagens que podem estar na fila de entrada e saída respetivamente.

Relativamente às plataforma MAS os resultados mostram, novamente, o JADE como a plataforma que apresenta melhores resultados em termos de latência. De destacar que o JADE apesar de apresentar latências muitos superiores às do RabbitMQ em quase todos os cenários, no cenário de maior carga apresenta valores muito próximos e até melhores que o RabbitMQ. O JIAC e o SPADE continuam a apresentar valores muito elevados de latência.

Em termos de escalabilidade na latência, em geral notamos grandes aumentos com o aumento do número de produtores e consumidores em todas as plataformas. Destaca-se o JADE que parece sofrer maior impacto com o aumento do número de consumidores, que com o aumento do número



	10 produtores 10 consumidores	10 produtores 50 consumidores	10 produtores 100 consumidores	50 produtores 10 consumidores	50 produtores 50 consumidores
ActiveMQ	30.85	63.69	70.21	192.32	896.00
Apache Kafka	604.11	3210.72	3977.41	3901.26	9556.63
ZeroMQ	2.59	236.50	425.33	904.42	1473.49
RabbitMQ	1.75	11.08	31.85	41.38	279.39
JADE	14.35	31.58	380.60	206.65	271.36
JIAC	1267.50	6096.21	18655.69	9155.64	18943.35
SPADE	1065.47	3369.97	6684.68	3768.07	9529.35

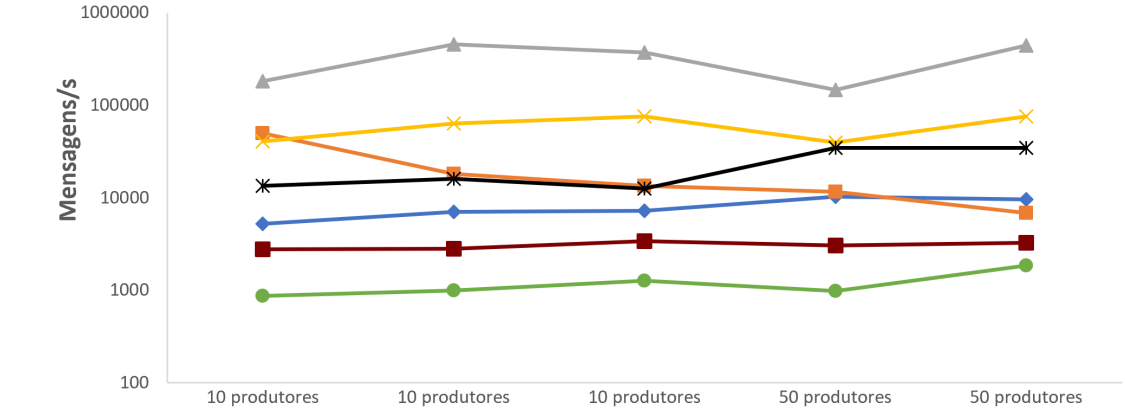
Figura 4.3: Latência na Troca de Mensagens - Um-para-Muitos - Sem Segurança

de produtores, ao contrário das restante plataformas.

O JADE neste tipo de comunicação apresentou valores de uso de memória bastante elevados. Estes valores andam próximos dos 4GB (juntando memória dos produtores e consumidores) para o cenário de 50 produtores e 50 consumidores, devido à necessidade de aumentar as filas usadas pelo JADE. Estas filas enchem devido à carga mais intensiva do número de mensagens enviadas e à incapacidade da entrega das mensagens suficientemente rápida. Este comportamento aconteceu de forma excecional com o JADE neste cenário, não detetamos comportamentos semelhantes para outro cenário ou *middleware*.

Em termos da taxa de transferência de mensagens, o vencedor continua a ser o ZeroMQ. Os valores que os restantes *middleware* apresentam são mais do que razoáveis pelo que não esperamos limitações nesse sentido (como pode ser visto na Fig. 4.4). O SPADE e o JADE continuam a apresentar valores bastantes razoáveis. De referir o aumento destes valores por parte do JADE e diminuição dos valores apresentados pelo SPADE neste tipo de comunicação. O JIAC apresenta valores bastante inferiores.

Em termos de escalabilidade na taxa de transferência de mensagens, destacar o ActiveMQ que parece ter atingido um máximo por volta das 10 mil mensagens por segundo e também o SPADE com um máximo por volta das 3 mil mensagens por segundo. O Apache Kafka, mais um vez, apresenta valores que diminuem com o aumento do número de produtores e consumidores (cenário contrário às restantes plataformas).



	10 produtores 10 consumidores	10 produtores 50 consumidores	10 produtores 100 consumidores	50 produtores 10 consumidores	50 produtores 50 consumidores
ActiveMQ	5216.48	7095.42	7259.32	10305.45	9582.80
Apache Kafka	50000.00	18154.09	13488.36	11639.84	6893.92
ZeroMQ	181159.40	454545.50	372023.80	146455.80	441889.33
RabbitMQ	40683.48	63516.26	75895.57	39494.47	76108.13
JADE	13466.20	16159.27	12633.82	34698.73	34882.10
JIAC	870.47	1003.68	1268.52	980.56	1865.67
SPADE	2790.64	2801.79	3397.65	3058.49	3261.54

Figura 4.4: Taxa de Transferência de Mensagens - Um-para-Muitos - Sem Segurança

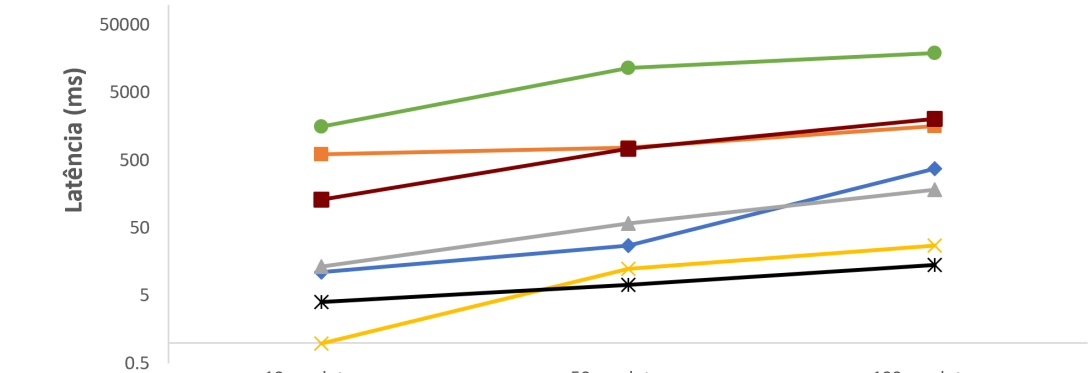
4.3.1.3 Muitos-para-Um

Os resultados dos testes de desempenho realizados mostram que, na comunicação muitos-para-um, é o JADE que apresenta melhores latências globalmente, como pode ser visto na Fig. 4.5. O RabbitMQ apresenta valores próximos, mas com desempenho superior no cenário com apenas 10 produtores. O ActiveMQ e o ZeroMQ apenas surgem depois com diferenças já bastante acentuadas. O JIAC e o SPADE mostram um comportamento muito aproximado em qualquer um dos cenários estudados, mantendo valores bastante elevados de latência.

Em termos de escalabilidade na latência, destacam-se o RabbitMQ, mas principalmente o JADE, por apresentarem uma escalabilidade bastante boa. Por outro lado, o ActiveMQ sofreu um aumento bastante acentuado na latência registada para o cenário de 100 produtores, revelando problemas de escalabilidade.

Em relação à taxa de transferência de mensagens o vencedor continua a ser o ZeroMQ, como pode ser visto na Fig. 4.6. Mais uma vez o ActiveMQ e o JIAC apresentam valores bastante mais baixos que os restantes. As restantes plataformas apresentam valores bastante razoáveis.

Em termos de escalabilidade na taxa de transferência de mensagens, destacar o RabbitMQ que foi aumentando bastante a sua taxa de transferência de mensagens com o aumento do número de agentes, revelando boa escalabilidade. O Apache Kafka apresenta uma diminuição da taxa de transferência de mensagens com o aumento do número de agentes, revelando má escalabilidade.



	10 produtores	50 produtores	100 produtores
ActiveMQ	11.15	27.60	380.99
Apache Kafka	616.65	768.00	1628.38
ZeroMQ	13.45	58.41	185.90
RabbitMQ	0.99	12.65	27.80
JADE	4.03	7.23	14.45
JIAC	1573.76	11747.28	19475.01
SPADE	132.97	740.86	2078.67

Figura 4.5: Latência na Troca de Mensagens - Muitos-para-Um - Sem Segurança

4.3.2 Testes Com Segurança

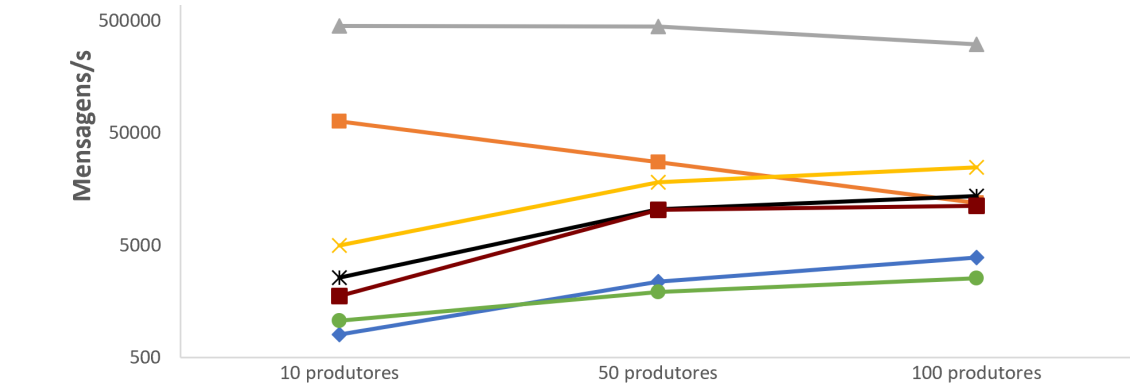
Os resultados dos testes de desempenho, agora com o uso de segurança, são mostrados aqui. Os mecanismos de segurança utilizados permitem garantir integridade, confidencialidade e autenticidade. Espera-se que estes resultados, quando comparados com os resultados de desempenho sem o uso de segurança, nos permitam perceber o impacto que os mecanismos de segurança trazem ao desempenho da comunicação.

O cliente C# de ZeroMQ usado na versão sem segurança, NetMQ 3.3.3.4, e recomendado pela documentação do ZeroMQ, apesar de ter recentemente recebido suporte para o uso de encriptação de curva elíptica, ainda não tem suporte para o uso de certificados na autenticação de um cliente. Assim, foi utilizado o clrzmq4 4.1.0.31 que serve todas as nossas necessidades quanto à segurança, sendo também referenciado pela documentação do ZeroMQ.

4.3.2.1 Um-para-Um

Os resultados dos testes de desempenho realizados mostram que, com o uso de segurança neste tipo de comunicação, há um decréscimo significativo nos valores de desempenho apresentados. Este decréscimo permitiu que os valores de desempenho do ZeroMQ e do RabbitMQ se aproximassem, mas o ZeroMQ continua a apresentar melhores resultados com o aumento do número de pares de agentes, como demonstra a Fig. 4.7.

O ActiveMQ, que estava próximo de ambos o ZeroMQ e o RabbitMQ na comunicação sem segurança, teve agora um aumento muito acentuado na latência apresentada. De notar também uma diminuição na latência do Apache Kafka, contrária ao que seria de esperar, mas que apesar de várias repetições dos testes se manteve. Acreditamos que esta melhoria das latências seja



	10 produtores	50 produtores	100 produtores
ActiveMQ	807.75	2365.41	3896.51
Apache Kafka	64102.56	27502.75	12033.69
ZeroMQ	454545.50	446428.60	310559.00
RabbitMQ	5005.01	18382.35	25012.51
JADE	2589.33	10548.52	13762.73
JIAC	1071.93	1917.26	2554.59
SPADE	1770.99	10341.74	11320.90

Figura 4.6: Taxa de Transferência de Mensagens - Muitos-para-Um - Sem Segurança

provocada por uma diminuição no congestionamento, dado que as mensagens são produzidas mais lentamente com o uso da segurança.

O desempenho relativo entre as plataformas multi-agente estudadas mantém-se, com o JADE a ser a que apresenta menores latências. O SPADE e o JIAC continuam a apresentar valores muito mais elevados. O aumento devido ao uso de segurança é pouco notório no SPADE e no JIAC por estes apresentarem valores de latência muito elevados que acabam por absorver parte desses aumentos, sendo bastante mais notório no JADE (por exemplo para o cenário de 100 pares de agentes temos um valor de latência quase 10 vezes superior ao registado sem o uso de segurança).

Em termos de escalabilidade na latência, notamos que o ZeroMQ e o RabbitMQ apresentam uma ótima escalabilidade. O JADE tem uma aumento bastante brusco, em valor absoluto, na passagem dos 50 para os 100 pares de agentes, pelo que é indicador que poderia não escalar para um maior número de agentes. O mesmo acontece com o ActiveMQ na passagem dos 10 para os 50 pares de agentes. As restante plataformas também apresentam fraca escalabilidade.

Em relação à taxa de transferência de mensagens, os resultados mostram que todos os *middleware* apresentam valores bastante razoáveis, como demonstra a Fig. 4.8. No entanto, é importante destacar o grande decréscimo na taxa de transferência de mensagens por parte do ZeroMQ, não só devido ao uso de segurança mas também possivelmente devido ao uso do novo cliente C#. Os resultados dos MAS mostram o JADE e o SPADE com valores razoáveis, mas o JIAC com valores bastante mais baixos.

Em termos de escalabilidade na taxa de transferência de mensagens, o RabbitMQ, o JADE e o SPADE apresentam escalabilidades bastante boas. Por outro lado, o ActiveMQ manteve a sua taxa em todos os cenários (aproximadamente nas 1900 mensagens por segundo), revelando má escalabilidade. O Apache Kafka, mais uma vez, apresenta má escalabilidade com uma taxa de

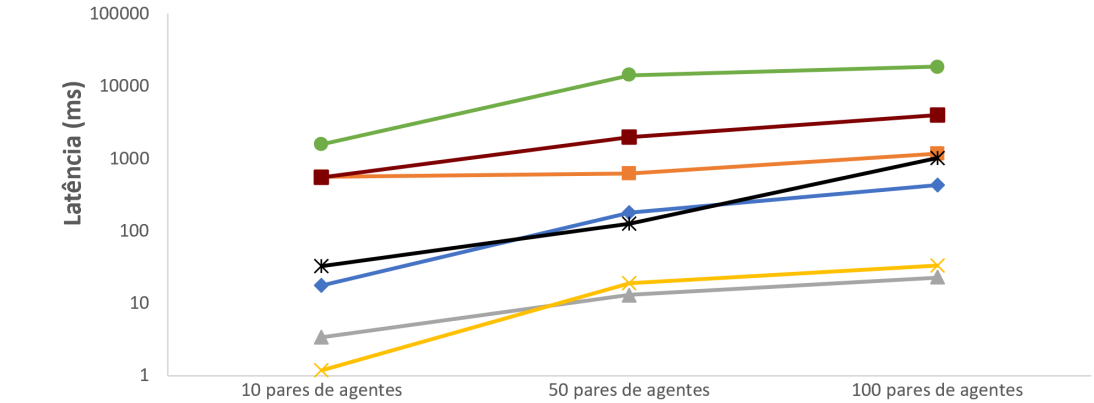


Figura 4.7: Latência na Troca de Mensagens - Um-para-Um - Com Segurança

transferência de mensagens que tende a diminuir com o aumento do número de agentes.

A Tabela 4.1 mostra o impacto do uso de segurança neste cenário de comunicação, quando comparado com a versão sem o uso de segurança.

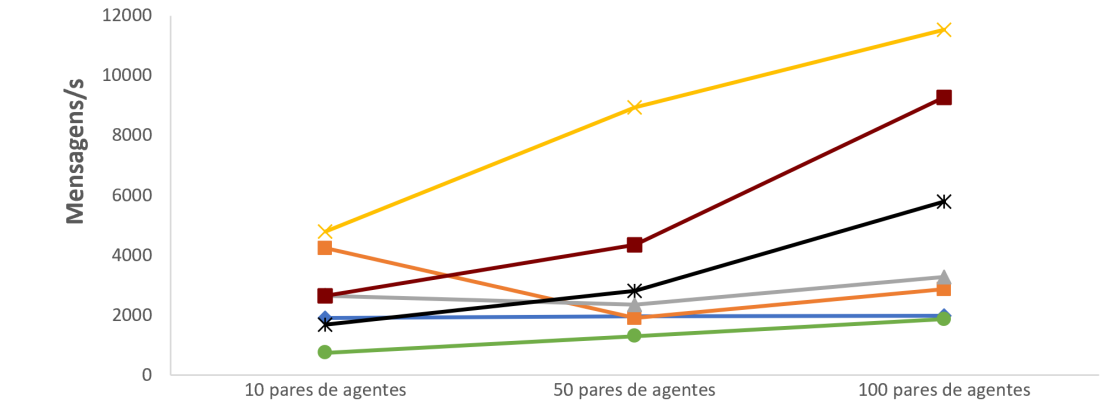
Tabela 4.1: Decréscimo de Desempenho com a Segurança - Um-para-Um

Métrica	**ActiveMQ**	**Apache Kafka**	**ZeroMQ**	**RabbitMQ**	**JADE**	**JIAC**	**SPADE**
Latência	953%	-16%	589%	37%	864%	23%	45%
Mensagens/s	-19%	-62%	-83%	-17%	-53%	-11%	-36%

4.3.2.2 Um-para-Muitos

Os resultados dos testes de desempenho realizados mostram que, neste tipo de comunicação, com o aumento do número de mensagens enviadas, é o RabbitMQ que apresenta menor latência, como demonstra a Fig. 4.9. Na comunicação *broadcast*, mais uma vez, o ZeroMQ apresenta valores de latência bastante superiores aos do RabbitMQ, estando o ActiveMQ e o JADE entre ambos. O Apache Kafka, o JIAC e o SPADE mais uma vez têm latências muito superiores. Destacar ainda um valor inesperado no ActiveMQ para o cenário de 10 produtores e 10 consumidores, em que se obteve um valor muito elevado e que depois desce nos cenários seguintes.

Tal como já tinha acontecido neste tipo de comunicação sem segurança, para evitar perda de mensagens no ZeroMQ foi necessário o aumento dos valores de *ReceiveHighWatermark* e de *SendHighWatermark*. Devido ao uso da segurança, que requer computação intensiva, foi agora



	10 pares de agentes	50 pares de agentes	100 pares de agentes
ActiveMQ	1904.76	1967.73	1981.38
Apache Kafka	4244.48	1901.43	2870.92
ZeroMQ	2656.75	2360.05	3281.92
RabbitMQ	4789.27	8950.95	11528.71
JADE	1678.98	2810.88	5792.40
JIAC	744.48	1301.43	1870.92
SPADE	2648.99	4360.05	9281.92

Figura 4.8: Taxa de Transferência de Mensagens - Um-para-Um - Com Segurança

também necessário aumentar o número de *input/output threads* usados (de raiz é usado apenas um *thread*, o que provocava perda de mensagens).

Em termos de escalabilidade na latência, tal como aconteceu neste cenário sem o uso de segurança, todas as plataformas apresentaram um grande aumento no valores de latência apresentados.

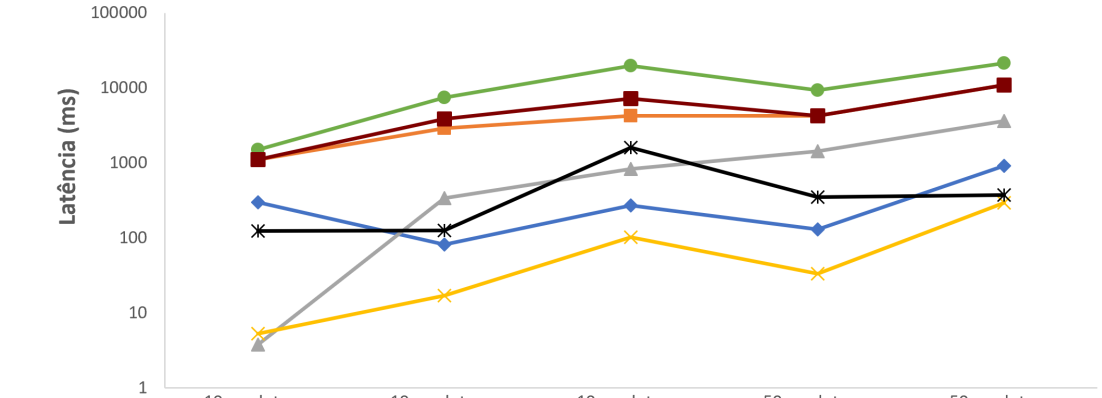
Em relação à taxa de transferência de mensagens, os resultados mostram que em geral as plataformas apresentam valores bastante razoáveis, como pode ser visto na Fig. 4.10. No entanto, o ActiveMQ, o SPADE e o principalmente o JIAC apresentam valores muito inferiores às restantes.

Em termos de escalabilidade na taxa de transferência de mensagens, o ZeroMQ é a plataforma que apresenta melhor escalabilidade. O ActiveMQ apresenta uma tendência de estabilizar nas 2500 mensagens por segundo indicando má escalabilidade. O mesmo acontece com o RabbitMQ entre as 45 e as 50 mil mensagens por segundo. O Apache Kafka, mais uma vez, apresenta má escalabilidade com uma taxa de transferência de mensagens que tende a diminuir com o aumento do número de agentes.

A Tabela 4.2 mostra o impacto do uso de segurança neste cenário de comunicação, quando comparado com a versão sem o uso de segurança.

Tabela 4.2: Decréscimo de Desempenho com a Segurança - Um-para-Muitos

Métrica	**ActiveMQ**	**Apache Kafka**	**ZeroMQ**	**RabbitMQ**	**JADE**	**JIAC**	**SPADE**
Latência	36%	11%	106%	24%	185%	10%	13%
Mensagens/s	-64%	-56%	-59%	-23%	-39%	-8%	-2%



	10 produtores 10 consumidores	10 produtores 50 consumidores	10 produtores 100 consumidores	50 produtores 10 consumidores	50 produtores 50 consumidores
ActiveMQ	299.74	82.23	273.01	130.18	921.00
Apache Kafka	1115.47	2897.79	4264.55	4294.01	11042.34
ZeroMQ	3.79	339.86	831.08	1428.95	3657.91
RabbitMQ	5.33	16.92	102.76	33.48	293.00
JADE	123.14	125.51	1602.71	349.87	374.87
JIAC	1501.88	7432.99	19673.65	9354.79	21364.77
SPADE	1115.47	3897.79	7264.55	4294.01	11042.34

Figura 4.9: Latência na Troca de Mensagens - Um-para-Muitos - Com Segurança

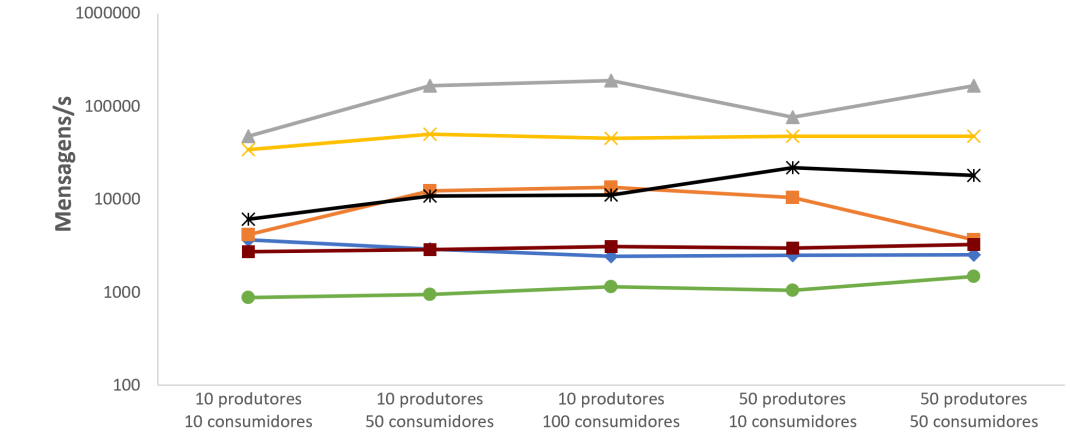
4.3.2.3 Muitos-para-Um

Os resultados dos testes de desempenho realizados mostram que, na comunicação muitos-para-um, o RabbitMQ é globalmente a plataforma que apresenta melhores valores de latência, como demonstra a Fig. 4.11. Seguem-se o ActiveMQ, o ZeroMQ e o JADE que apresentam valores próximos entre si mas já bastante superiores aos do RabbitMQ. No cenário de 100 produtores o ActiveMQ sofre um aumento bastante acentuado, mas apesar de várias repetições dos testes os valores mantiveram-se. Os restantes *middleware* e MAS apresentam valores bastante mais elevados. Relativamente ao testes sem segurança assistimos a um aumento bastante acentuado por parte do JADE que foi assim ultrapassado pelo RabbitMQ.

Em termos de escalabilidade na latência, destacam-se o RabbitMQ e o JADE por apresentarem uma escalabilidade razoável quando comparada com as restantes plataformas. O ActiveMQ sofreu um aumento bastante acentuado na latência registada para o cenário de 100 produtores, tal como já tinha acontecido sem o uso de segurança.

Em relação à taxa de transferência de mensagens, os resultados mostram que todos, exceto o JIAC e o ActiveMQ, apresentam valores razoáveis, como demonstra a Fig. 4.12. No entanto, e de acordo com os testes já apresentados anteriormente, a taxa de transferência de mensagens é bastante superior para o ZeroMQ e o RabbitMQ.

Em termos de escalabilidade na taxa de transferência de mensagens, destacar o RabbitMQ, o ZeroMQ e o JADE que foram aumentando bastante a sua taxa de transferência de mensagens com o aumento do número de agentes, revelando boa escalabilidade. O Apache Kafka, mais um vez, apresenta uma diminuição da taxa de transferência de mensagens com o aumento do número



	10 produtores 10 consumidores	10 produtores 50 consumidores	10 produtores 100 consumidores	50 produtores 10 consumidores	50 produtores 50 consumidores
ActiveMQ	3659.25	2926.78	2457.68	2504.41	2552.00
Apache Kafka	4162.50	12302.54	13488.36	10493.18	3698.72
ZeroMQ	47755.49	166777.90	190985.50	76616.61	167515.41
RabbitMQ	34435.26	50648.30	45741.47	48049.20	48004.92
JADE	6153.85	10939.96	11120.75	21987.69	18207.24
JIAC	884.36	953.69	1155.12	1059.87	1483.56
SPADE	2759.66	2880.18	3135.19	2992.22	3269.48

Figura 4.10: Taxa de Transferência de Mensagens - Um-para-Muitos - Com Segurança

de agentes, embora menos acentuada, mas revelando má escalabilidade. As restantes plataformas apresentam uma escalabilidade razoável.

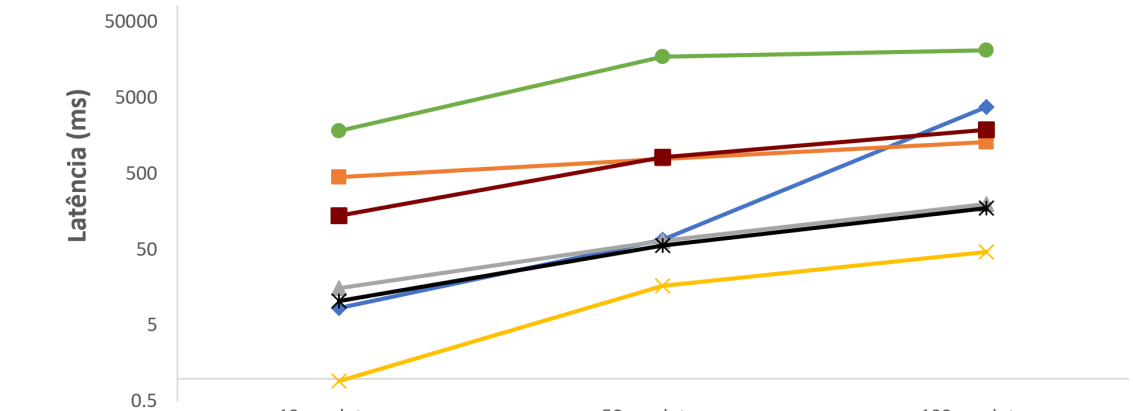
A Tabela 4.3 mostra o impacto do uso de segurança neste cenário de comunicação, quando comparado com a versão sem o uso de segurança. De notar uma diminuição na latência do Apache Kafka e do SPADE, provavelmente provocada por uma diminuição no congestionamento, dado que as mensagens são produzidas mais lentamente com o uso da segurança.

Tabela 4.3: Decréscimo de Desempenho com a Segurança - Muitos-para-Um

Métrica	**ActiveMQ**	**Apache Kafka**	**ZeroMQ**	**RabbitMQ**	**JADE**	**JIAC**	**SPADE**
Latência	820%	-15%	9%	55%	847%	23%	-3%
Mensagens/s	-41%	-85%	-97%	-35%	-48%	-15%	-29%

4.3.3 Conclusões dos Testes de Desempenho Realizados

Os resultados dos testes sem o uso de segurança mostram o ZeroMQ como o *middleware* que apresenta a melhor taxa de mensagens enviadas em todos os cenários e com uma vantagem bastante expressiva. Em relação à latência, os resultados mostram o ZeroMQ como o melhor na comunicação um-para-um, o RabbitMQ como o melhor na comunicação um-para-muitos e o JADE na comunicação muitos-para-um. O RabbitMQ, apesar de ser o melhor em termos de latência apenas em *broadcast*, foi a plataforma que apresentou os melhores resultados globalmente (2ª melhor nos restantes cenários). O mesmo acontece para a taxa de transferência de mensagens em que apesar da superioridade do ZeroMQ em todos os cenários, o RabbitMQ sempre apresentou



	10 produtores	50 produtores	100 produtores
ActiveMQ	8.55	67.45	3787.05
Apache Kafka	447.51	782.26	1319.24
ZeroMQ	15.55	65.26	200.51
RabbitMQ	0.93	16.67	46.70
JADE	10.54	57.14	175.70
JIAC	1818.20	17361.36	21096.27
SPADE	140.01	830.28	1906.97

Figura 4.11: Latência na Troca de Mensagens - Muitos-para-Um - Com Segurança

valores bastante elevados. Assim, concluímos que para a comunicação sem segurança a melhor plataforma é o RabbitMQ.

Nos testes com o uso de segurança, a taxa de transferência de mensagens do ZeroMQ aproximou-se dos restantes *middleware* (principalmente do RabbitMQ), embora seja ainda maioritariamente superior. Quanto aos valores de latência, os resultados mostram o RabbitMQ como a plataforma que apresenta melhor desempenho na comunicação um-para-muitos e agora também na comunicação muitos-para-um. Na comunicação um-para-um, o ZeroMQ foi novamente superior com o aumento do número de pares de agentes, no entanto, o RabbitMQ está agora bastante mais próximo. Assim concluímos que, também para a comunicação com segurança, a melhor plataforma é o RabbitMQ, tendo em conta que este manteve-se superior onde já o era, aproximou-se do ZeroMQ nos pontos onde era inferior e ultrapassou o JADE na comunicação muitos-para-um. O uso de segurança na comunicação teve um impacto no desempenho global das plataformas, resultando num aumento médio da latência de: 23.11% nos *middleware* e 17.12% nos MAS. E numa diminuição na taxa de transferência de mensagens de: 67.85% nos *middleware* e 37.10% nos MAS. No entanto, estes valores percentuais não traduzem o efeito real que a segurança teve nas plataformas. Os *middleware* apresentam um maior impacto percentual devido ao uso de segurança porque as plataformas MAS apresentam, na maioria dos cenários, valores de latência muito superiores pelo que o impacto da segurança é depois muito absorvido, resultando em aumentos percentuais mais baixos. Da mesma forma, os *middleware* apresentam, na maioria do cenários, taxas de mensagens enviadas bastante superiores que têm assim mais tendência a descer com o uso de segurança, o que se reflete em decréscimos percentuais superiores.

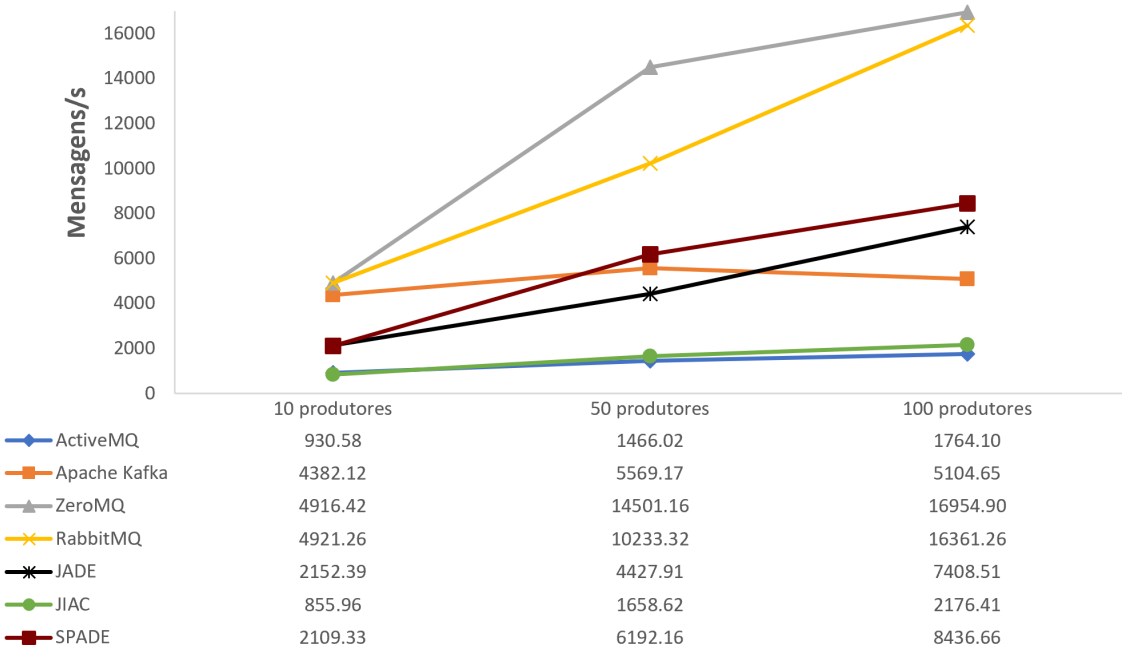


Figura 4.12: Taxa de Transferência de Mensagens - Muitos-para-Um - Com Segurança

Quanto às restantes plataformas, que foram sendo menos referidas por apresentarem resultados mais fracos, seria importante referir que o ActiveMQ apesar de apresentar alguns bons resultados em termos de latência, nunca apresentou um desempenho suficientemente bom para ultrapassar ambos o ZeroMQ e o RabbitMQ. Para além disso, a taxa de transferência de mensagens que apresentou foi maioritariamente bastante baixa levando-nos a descartar este *middleware*. O Apache Kafka apresentou sempre valores de latência muito elevados que, apesar das tentativas de resolução usando diferentes configurações do broker bem como diferentes clientes C#, não conseguimos resolver. Por este motivo descartamos este *middleware*. O JADE é, sem qualquer dúvida, a plataforma multi-agente estudada que apresenta melhor desempenho com e sem o uso de segurança na comunicação, aproximando-se e até ultrapassando os *middleware* em alguns cenários. O JIAC apesar do uso do ActiveMQ como *middleware* de comunicação apresentou resultados muito fracos. As latências foram sempre muito elevadas o que não é coerente com o desempenho apresentado pelo ActiveMQ isolado, possivelmente algum mecanismo interno da plataforma será responsável por estes valores. Os valores de mensagens enviadas por unidade de tempo, apesar de mais fracos que o esperado, podem ter alguma relação com o próprio ActiveMQ que isolado também apresentou valores baixos. O SPADE também apresentou latências muito elevadas. No que diz respeito às mensagens enviadas por unidade de tempo apresentou um desempenho bastante razoável e próximo daquele que o JADE apresenta.

A Tabela 4.4 fornece algumas estatísticas globais que permitem melhor compreender os resultados obtidos e as razões da nossa escolha. Os valores apresentados refletem o número de vezes que cada plataforma foi a melhor (linhas 1 e 2) e o número de vezes que cada plataforma ficou

nas melhores três (linhas 3 e 4). Os resultados são apresentados no formato latência / taxa de transferência de mensagens. As duas últimas linhas mostram o impacto da segurança na latência e na taxa de transferência de mensagens, respetivamente.

Tabela 4.4: Comparação Global das Plataformas de Comunicação Estudadas

Métrica	ActiveMQ	Apache Kafka	ZeroMQ	RabbitMQ	JADE	JIAC	SPADE
1º Inseguro	0 / 0	0 / 1	3 / 10	5 / 0	3 / 0	0 / 0	0 / 0
1º Seguro	0 / 0	0 / 0	3 / 7	8 / 4	0 / 0	0 / 0	0 / 0
Top 3 Inseguro	7 / 0	0 / 6	5 / 11	11 / 11	10 / 3	0 / 0	0 / 2
Top 3 Seguro	7 / 0	0 / 5	7 / 11	11 / 11	8 / 2	0 / 0	0 / 4
Latência	603%	-7%	235%	39%	632%	18%	19%
Mensagens/s	-41%	-68%	-80%	-25%	-47%	-11%	-22%

A tabela mostra que os melhores resultados foram sempre obtidos por três plataformas: o ZeroMQ, o RabbitMQ e o JADE. Em relação ao top 3 conseguimos ver que as três plataformas continuam a ser as que apresentam melhores resultados. No entanto, destaca-se o RabbitMQ por estar no top 3 em todos os cenários analisados. A tabela mostra ainda o impacto médio no desempenho que a segurança causou em cada uma das plataformas. O RabbitMQ foi a plataforma que sofreu menos impacto com o uso de segurança (exceto quando comparado com plataformas que tinham valores muito elevados e que absorveram o impacto). Os resultados já descritos acima demonstram exatamente esta situação, com o RabbitMQ a ganhar vantagem nos cenários com o uso de segurança.

Com esta análise concluímos que a melhor plataforma estudada é o RabbitMQ e será esta a melhor opção para integração como *middleware* numa plataforma multi-agente. O RabbitMQ apresentou melhor latência em quase todos os cenários, e apesar de não ser o *middleware* que apresenta melhores taxas de transferência de mensagens, os valores que apresentou são sempre bastante elevados e mais do que suficientes para o uso na maioria das plataformas. Foi também o mais consistente ao longo dos testes, bem como aquele que sofreu menor impacto com o uso de segurança (cenário em que a arquitetura a desenvolver se foca). Para além desta análise, puramente baseada em resultados, a configuração e o uso de todos as plataformas permitiu perceber que o RabbitMQ é a plataforma que apresenta melhor documentação e melhor API de trabalho.

4.4 Proposta de Solução

Os resultados dos testes realizados mostram a melhoria de desempenho que os *middleware* orientados a mensagens podem trazer às plataformas multi-agente em geral. Pensando no caso específico da plataforma do LIACC, a única desvantagem da integração de um novo *middleware* de comunicação no AgentService é a não substituição completa do AgentService. O uso de uma plataforma MAS que, ao contrário do AgentService, apresente desenvolvimento e suporte ativo pode ser benéfico no sentido em que seria possível receber as mais recentes atualizações, como

melhorias de desempenho e correção de problemas. No entanto, como o foco desta dissertação é a comunicação segura e eficiente consideramos que essa substituição pode ser um trabalho futuro se assim se justificar. Até ao momento não identificamos problemas nas funcionalidades que o AgentService está a desempenhar, exceto na troca de mensagens.

A plataforma do LIACC é constituída por um conjunto heterogéneo de agentes com as mais diversas funções. Estes agentes devem utilizar um *middleware* de comunicação para a troca de mensagens entre eles. Assim, seguindo o modelo de referência para o transporte de mensagens da FIPA [FIPA TC Agent Management, 2002], propomos a arquitetura da Fig. 4.13. Esta arquitetura introduz o RabbitMQ como serviço de transporte de mensagens (MTS) entre agentes, mas também para os serviços de páginas amarelas (DF) e páginas brancas (AMS). No entanto, todo o processamento do DF e AMS continua a ser realizado pelo AgentService. O uso do RabbitMQ tem como objetivo a realização destas operações de formas segura, funcionando como um *wrapper*.

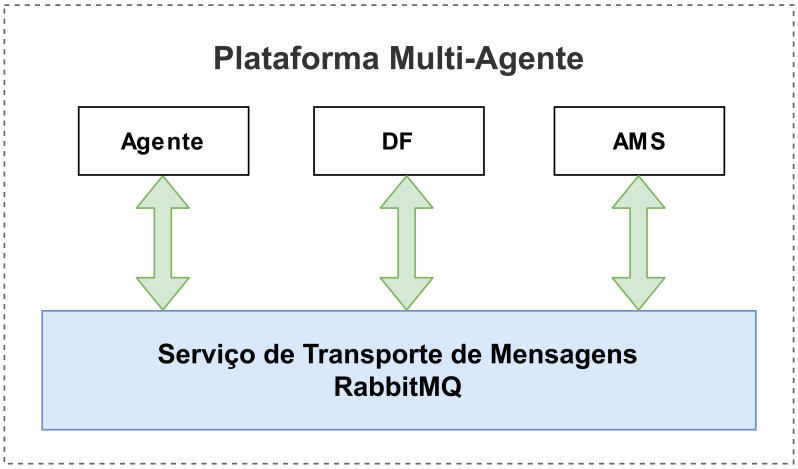


Figura 4.13: Arquitetura Geral da Solução

A introdução de mecanismos de segurança na plataforma, mesmo que já integrados no RabbitMQ, implicam que cada agente da plataforma tenha agora um par de chaves (chave pública e chave privada) e o respetivo certificado. O DF e o AMS podem também ser vistos como agentes da plataforma pelo que ambos devem ter um par de chaves e o respetivo certificado. Assim, propomos a criação de uma autoridade de certificação que irá emitir certificados para os agentes, permitindo assegurar a autenticidade, confidencialidade e integridade desejadas.

As plataformas multi-agente podem ser usadas em cenários que requerem muito poder computacional, surgindo a necessidade de distribuir os agentes por vários computadores [Abbas and Egerstedt, 2011]. A arquitetura destas plataformas é inerentemente distribuída, permitindo a distribuição dos agentes por diversas máquinas, com uma ou várias instâncias da plataforma MAS (usando o mecanismo de federação). No entanto, estes sistemas estão apenas disponível para agentes *in-platform*, isto é, agentes criados diretamente na plataforma e não para o uso com agentes externos. Os agentes externos permitem que uma plataforma multi-agente seja um componente de um sistema maior e não todo o sistema (em que tudo seria construído dentro do MAS).

A plataforma do LIACC, ao distribuir os seus agentes/aplicações por diversas máquinas, consegue distribuir todo o trabalho realizado por cada agente/aplicação, permitindo realizar simulações com cenários de maior dimensão e complexidade, com mais agentes e equipas. Apesar de ter sido desenhada com esse objetivo, a plataforma do LIACC não tinha ainda suporte para fazer esta distribuição de forma automática. Para além disso, o desenvolvimento de uma arquitetura segura para a comunicação entre agentes, descrita na secção 5.2, deveria considerar a possibilidade dos agentes correrem em diferentes máquinas, mantendo a segurança do sistema. A plataforma do LIACC, e outras plataformas do mesmo género, podem usar este sistema de distribuição para as suas aplicações e tirar benefícios disso [The AgentService Team, 2008].

Assim sugerimos um novo tipo de componente para a plataforma, o Nó de Simulação. Um Nó de Simulação consiste numa aplicação que recebe tarefas de execução de agentes, sob a forma de mensagens. A transmissão de tarefas ao Nó de Simulação é feita usando o RabbitMQ de forma segura, o que lhe permite saber quem lhe fez o pedido e reagir em função disso. A arquitetura interna consiste num processo à escuta de mensagens, que para cada mensagem recebida lança um *thread* que a irá processar (funcionando como um *dispatcher*). O *thread* responsável por uma dada mensagem irá processá-la, podendo descartá-la quando deteta alguma incoerência ou então lançar o agente respetivo em execução. Um Nó de Simulação corre numa máquina disponível para a execução de agentes, sendo que a plataforma pode ter várias dessas máquinas e, por isso, vários Nós de Simulação preparados para a execução de agentes. Deve existir um outro componente que é responsável pelo envio das mensagens de lançamento de agentes para os Nós de Simulação respetivos, neste caso, o Painel de Controlo.

A Figura 4.14 mostra a arquitetura lógica de lançamento de agentes pelo Painel de Controlo, usada pela plataforma do LIACC. Antes da realização desta dissertação, este lançamento apenas podia ser feito de forma local (na mesma máquina que o Painel de Controlo) ou então de forma manual numa outra máquina. A arquitetura física de lançamento e distribuição de agentes que propomos na Fig. 4.15 permite realizar a distribuição de forma automática usando os Nós de Simulação. O Painel de Controlo envia um pedido de lançamento de um agente para um dado Nó de Simulação (via RabbitMQ) e este lança, localmente, o agente em execução.

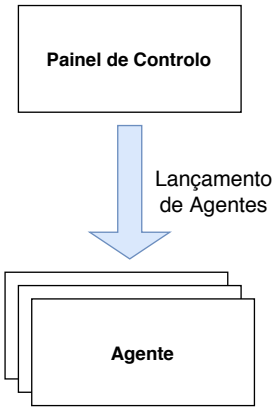


Figura 4.14: Arquitetura Lógica de Distribuição de Agentes

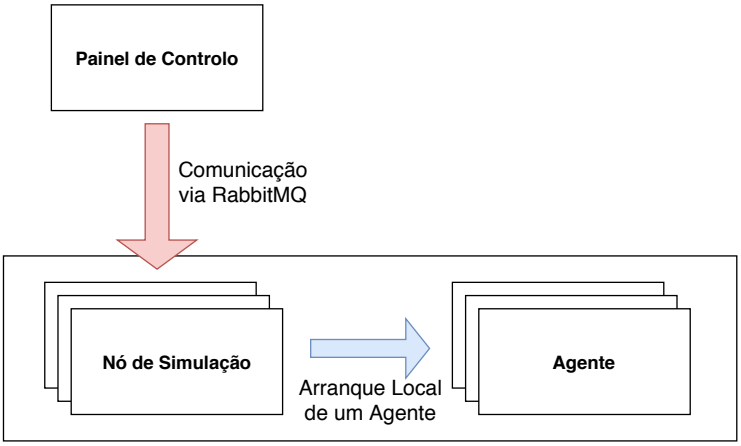


Figura 4.15: Arquitetura Física de Distribuição de Agentes

De forma a instanciar a arquitetura proposta na plataforma do LIACC, é necessária a realização de três tarefas principais: substituição do AgentService pelo RabbitMQ, introdução de comunicação segura na plataforma e a distribuição dos agentes externos por diversas máquinas.

O primeiro passo deve ser o estudo da plataforma do LIACC, de forma a perceber o seu funcionamento interno mais em detalhe e a interação entre os diferentes componentes. Depois, podemos iniciar o processo de substituição do AgentService pelo RabbitMQ, criando um protótipo funcional da plataforma com este novo *middleware*. No final desta primeira fase, todos os componentes da plataforma do LIACC devem funcionar, comunicando com recurso ao RabbitMQ. De seguida, pode ser realizada a adaptação da plataforma para o uso de comunicação segura com o RabbitMQ. Este processo implica, maioritariamente, o desenvolvimento de infraestruturas que permitam a cada agente ter um certificado para garantir a sua identidade, nomeadamente a criação da autoridade de certificação e todo o mecanismo envolvente. Para além disso, é necessária a criação de suporte para o uso de diferentes níveis de segurança. Por fim, o suporte para distribuição dos agentes externos por diversas máquinas implica a criação dos Nós de Simulação, bem como a gestão da comunicação entre o Painel de Controlo e os Nós de Simulação (incluindo o suporte para lançamento remoto destes). Para além disso, são necessárias várias adaptações à plataforma para manter o funcionamento correto de todo o sistema. No capítulo 5 é detalhado todo este processo de adaptação e integração da solução na plataforma do LIACC, bem como todos os desafios que surgiram ao longo do processo. Esta descrição detalhada do processo de adaptação e integração deverá ser útil para a integração desta arquitetura genérica noutras plataformas multi-agente.

A solução desenvolvida será avaliada no capítulo 6, através de testes de desempenho, de forma a perceber que melhorias conseguiu trazer à plataforma do LIACC. Estes testes foram realizados nos mesmos cenários dos testes às plataformas, isto é, comunicação um-para-um, um-para-muitos e muitos-para-um. Para além disso, a plataforma foi avaliada para a comunicação entre o *Disturbances Manager* e os agentes veículo, definido como o principal *bootleneck* da plataforma em [Almeida, 2017]. Esta avaliação da solução desenvolvida deverá permitir perceber se a integração do RabbitMQ numa plataforma MAS consegue assegurar comunicação mais eficiente e segura.

Capítulo 5

Especificação e Desenvolvimento

Este capítulo descreve todo o processo de implementação da solução na plataforma do LIACC. Está dividido em três grupos principais - a substituição do AgentService pelo RabbitMQ, a integração de comunicação segura na plataforma usando o RabbitMQ e a distribuição dos agentes externos da plataforma.

5.1 Substituição do AgentService pelo RabbitMQ

Um dos principais objetivos desta dissertação era a introdução de um *middleware* orientado a eventos responsável por toda a comunicação da plataforma. Assim, o primeiro passo do desenvolvimento foi a substituição do *middleware* do AgentService pelo novo *middleware*, o RabbitMQ. A introdução do RabbitMQ implicou a identificação de todos os módulos da plataforma que utilizavam comunicação, tendo sido identificados cinco: Painel de Controlo, agentes ATC, agentes Veículo, *Disturbances Manager* e Ferramenta de Monitorização. As alterações realizadas foram aproximadamente similares em cada um dos módulos/aplicações.

O uso do RabbitMQ na plataforma implica a instalação do *broker*/servidor, neste caso foi usada a versão 3.8.2, numa máquina disponível para o efeito. A plataforma apenas precisa do endereço IP dessa máquina para que os agentes se consigam ligar. Neste primeiro passo do desenvolvimento foram mantidas todas as configurações padrão do *broker* do RabbitMQ. Para além disso, com o novo componente da arquitetura, o *broker* do RabbitMQ, é agora necessário que aquando do arranque do Painel de Controlo (com o objetivo de iniciar um simulação), sejam iniciados ambos o AgentService e o servidor do RabbitMQ.

5.1.1 Cenários de Comunicação da Plataforma do LIACC

A introdução do RabbitMQ não foi realizada de forma direta, isto é, apenas substituindo todas as chamadas ao *middleware* do AgentService por chamadas ao RabbitMQ. Como o RabbitMQ aumenta as possibilidades da comunicação, nomeadamente com a introdução da capacidade de *broadcast* explicito, foram identificados cenários de comunicação que são uma mais valia para

a plataforma. Os cenários identificados foram implementados e integrados na plataforma, sendo descritos abaixo.

- O **cenário individual** usado nas comunicações um-para-um. Cada agente tem a sua própria fila de mensagens onde outros agentes podem publicar mensagens. A fila tem um nome no formato "Agent-<IdAgente>-MessageQueue"em que <IdAgente> corresponde ao identificador do agente (AID) a que pertence a fila. Essa fila está associada a um *exchange* do tipo *Direct* (chamado "DirectExchange"). Assim, as mensagens com *routing key* igual ao identificador do agente são enviadas para a fila de mensagens do agente. Um agente emissor só precisa de saber o identificador do agente para quem pretende enviar a mensagem, que pode ser encontrado com o uso das páginas amarelas e páginas brancas.

- O **cenário de equipa** usado nas comunicações *broadcast* para todos os elementos de uma equipa. Neste cenário são usados *exchange* do tipo *Fanout* com um *exchange* por equipa. Um *exchange* chama-se "FanoutExchange<IdEquipa>". Um agente apenas precisa de enviar uma mensagem para o *exchange* correto usando o identificador da equipa para a qual quer enviar a mensagem. No nosso caso, como queremos que cada elemento da equipa receba a mensagem, é necessário que cada agente da equipa tenha uma fila de mensagens ligada ao *exchange* da sua equipa. O nome da fila segue o formato "Agent-<IdAgente>-Team-<IdEquipa>-MessageQueue". O comportamento do RabbitMQ apenas usando uma fila seria distribuir uniformemente as mensagens por todos os agentes/consumidores.

  Na solução anterior, com o AgentService, o envio de uma mensagem para os agentes de uma equipa implicava a procura nas páginas amarelas/brancas por todos os agentes que pertenciam a essa equipa (usando o identificador da equipa) e depois o envio da mensagem individualmente para cada um deles.

- O **cenário de tipo de veículo** usado nas comunicações *broadcast* para todos os veículos de um determinado tipo (por exemplo para todos os aviões). Neste cenário são usados *exchange* do tipo *Fanout* com um *exchange* por tipo de veículo no formato "FanoutExchange<TipoVeículo>". Um agente apenas precisa de enviar uma mensagem para o *exchange* correto usando o identificador do tipo de veículo para o qual quer enviar a mensagem. Cada agente veículo de um determinado tipo tem uma fila de mensagens no formato "Agent-<IdAgente>-<TipoVeículo>-MessageQueue".

  Na solução anterior, com o AgentService, o envio de uma mensagem para os agentes de um determinado tipo de veículo não era possível.

- O **cenário de tópico** usado nas comunicações *broadcast* para todos os agentes interessados num determinado tópico. Este cenário, mais genérico, é semelhante a um cenário de multicast em que agentes criam tópicos em que outros podem estar interessados. Neste cenário são usados *exchange* do tipo *Fanout* com um *exchange* por tópico no formato "FanoutExchange<Tópico>". Um agente que pretenda publicar num determinado tópico apenas precisa de enviar uma mensagem para o *exchange* correto usando o identificador do

tópico. Um agente subscritor tem uma fila de mensagens no formato "Agent-<IdAgente>-<Topico>-MessageQueue"onde recebe mensagens publicadas no tópico.

Na solução anterior, com o AgentService, esta solução genérica não era possível.

Estes cenários podem ser divididos num cenário de comunicação individual e três cenários de *broadcast*. O uso destes cenários está adaptado à plataforma do LIACC, mas o cenário de comunicação individual e de tópico broadcast poderiam ser aplicados em qualquer MAS cobrindo os cenários típicos destes sistemas [Paletta, 2012].

5.1.2 Serialização de Mensagens

O RabbitMQ envia mensagens em formato binário. Para o envio de mensagens com formatos mais complexos, normalmente modeladas por objetos, é necessário o uso de serialização. O processo de serialização consiste na transformação de um objeto numa representação em memória (bytes), que pode ser usada para mais tarde o reconstruir [Wenzel et al., 2020]. Esta representação em memória pode ter formato XML, JSON ou binário em função da ferramenta usada.

O desempenho do RabbitMQ é influenciado pelo tamanho das mensagens a enviar, influenciando também o desempenho da plataforma. Para além deste atraso causado por mensagens com tamanhos superiores, podemos também considerar o atraso introduzido pelo próprio processo de serialização. Os tempos de serialização influenciam a latência entre a necessidade do envio de uma mensagem, por um determinado agente, e a sua chegada ao agente destinatário. Assim surge a necessidade de encontrar e usar uma ferramenta de serialização que seja eficiente em termos de tempo e espaço.

A serialização para XML e JSON foi a primeira a surgir devido à facilidade com que é interpretada por humanos (ajuda na procura de problemas na comunicação) e ao seu uso em aplicações Web. O XML tende a ocupar mais espaço que o JSON, mas ambos ocupam normalmente mais espaço do que a serialização binária. Para além disso, a necessidade de serem interpretados carácter a carácter torna também o processo mais lento [Mesfin et al., 2018]. Assim, para obter melhor desempenho na plataforma e porque não existe a necessidade das mensagens conseguirem ser interpretadas, optamos por ferramentas de serialização binária. A literatura referencia várias ferramentas como o Google's Protocol Buffers (ProtoBuf[1]), o Apache Thrift[2], o Apache Avro[3] e o MessagePack[4] [Giaimo et al., 2015, Eddelbuettel et al., 2016, Zhao et al., 2016].

De forma a perceber qual a que apresenta melhor desempenho, em termos de espaço e tempo para o uso na plataforma, decidimos realizar alguns testes de desempenho. Os testes consistem na serialização de um objeto que representa uma mensagem trocada entre agentes na plataforma. Foram realizados para as ferramentas mais referenciadas pela literatura e para a ferramenta de serialização binária padrão do C#. A Tabela 5.1 apresenta os resultados obtidos.

[1]Mais informação disponível em https://developers.google.com/protocol-buffers
[2]Mais informação disponível em http://thrift.apache.org/
[3]Mais informação disponível em https://avro.apache.org/
[4]Mais informação disponível em https://msgpack.org/

Tabela 5.1: Comparação do Desempenho das Ferramentas de Serialização

	.Net Bin.Formatter	MessagePack	ProtoBuf	Apache Avro	Apache Thrift
Serialização (ms)	0.25	0	0	0	3
Desserialização (ms)	0.25	0	0	0	0
Tamanho (Bytes)	4435	347	228	268	441

Os resultados mostram que todas as ferramentas apresentam bons tempos de serialização e desserialização, com exceção para o Apache Thrift que apresenta um tempo de serialização bastante superior. Por outro lado em termos de espaço notam-se maiores oscilações. O resultado da serialização com a ferramenta do .Net apresenta um tamanho mais de 10 vezes superior à ferramenta que se segue (Apache Thrift). Concluímos que a ferramenta que se adequa melhor aos objetos usados para a troca de mensagens pela plataforma é o Protobuf, por ser a que apresenta melhores resultados em termos de espaço. Assim, foi essa a ferramenta utilizada.

5.2 Comunicação Segura com o RabbitMQ

A comunicação segura era outro importante objetivo desta dissertação. Assim, o passo seguinte foi a introdução de segurança na comunicação entre agentes.

5.2.1 Arquitetura de Certificação de Agentes

Para o uso de comunicação segura no RabbitMQ é necessário que cada agente tenha acesso a um certificado em que o *broker* confia, isto é, um certificado emitido por uma autoridade de certificação (CA) que o *broker* conhece e confia. Assim uma das primeiras necessidades que tivemos foi a criação de uma autoridade de certificação para a plataforma. Todos os agentes irão confiar nesta autoridade de certificação e todos os agentes devem ter um certificado emitido por esta. Esta autoridade de certificação foi colocada no Painel de Controlo da plataforma sob a forma de um serviço. A CA precisa de um par de chaves (chave privada e chave pública). Todos os nós do sistema devem confiar na CA, pelo que é necessário que o certificado desta seja adicionado como confiável em todos os nós do sistema. No contexto da plataforma do LIACC, e de forma a facilitar o seu uso, são fornecidos dois *scripts* que permitem realizar os dois passos anteriores através do uso de chamadas à ferramenta OpenSSL[5].

Para além do par de chaves da CA, também alguns outros componentes têm certificados estáticos gerados utilizando *scripts*. O Painel de Controlo, o *broker* do RabbitMQ, os Nós de Simulação e a ferramenta de monitorização têm cada um o seu *script* individual para facilitar o processo. Na nossa implementação apenas o Nó de Simulação pode ter mais do que uma instância, mas o número de Nós de Simulação é controlado pelo operador do Painel de Controlo, pelo que cresce de forma controlada. Assim, em termos de escalabilidade, a pré-condição de geração dos certificados manualmente para estes componentes é facilmente praticada.

[5]Mais informação disponível em https://www.openssl.org/

Nesta fase foi necessária a introdução de algumas configurações no *broker* do RabbitMQ relativamente ao uso de segurança. Foi necessário especificar o caminho para o certificado a usar pelo *broker*, bem como o caminho para o certificado da CA do sistema. Depois, e de forma a garantir maior segurança, foram usadas as seguintes opções do SSL/TLS: opção *fail_if_no_peer_cert* como *true*, indicando a necessidade de um cliente apresentar um certificado perante o *broker*; opção *verify* como *verify_peer*, indicando a necessidade de verificar que o certificado apresentado pelo cliente é válido, segundo a CA do sistema; e a opção *versions* como "tlsv1.2"para forçar o uso da versão 1.2 do TLS, dado que o RabbitMQ não apresenta ainda suporte para a versão 1.3.

Quanto à geração de certificados dinâmicos, para agentes criados durante uma simulação, foi necessária a criação de uma arquitetura que o permitisse. A arquitetura de certificação criada, que pode ser vista na Fig. 5.1, consiste no seguinte conjunto de passos:

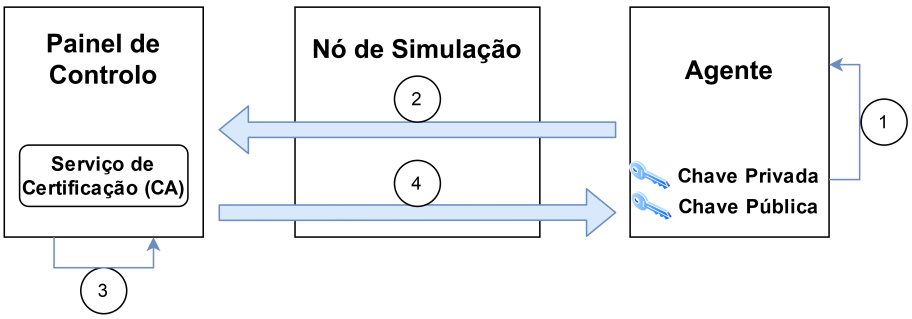


Figura 5.1: Arquitetura de Certificação de Agentes

1. Cada agente aquando da sua criação é responsável pela geração de um par de chaves, usadas posteriormente para assegurar a sua identidade. Este par é constituído por chave pública e chave privada.

2. De seguida, cada agente envia a sua chave pública para o Painel de Controlo para a criação de um certificado válido emitido pela CA do sistema, através de um serviço que o Painel de Controlo fornece. Um agente irá fazer o pedido de certificação por intermédio do Nó de Simulação que o criou. Assim, o serviço de certificação do sistema pode descartar pedidos que não venham de Nós de Simulação do sistema. Para além disso, como ambos o Painel de Controlo e o Nó de Simulação têm certificados já emitidos, esta comunicação pode ser realizada de forma segura usando RabbitMQ. O objetivo é garantir que apenas agentes que foram criados por Nós de Simulação do sistema (em que se confia) têm um certificado emitido e, assim, acesso à plataforma.

3. A CA no Painel de Controlo assina cada um dos certificados que recebe, verificando que vêm de um Nó de Simulação. Para além disso, é verificado que o identificador para o qual se vai gerar certificado não teve já um certificado gerado em seu nome.

4. O certificado é depois devolvido ao agente, que o pode usar em comunicações futuras com outros agentes da plataforma, que confiam na identidade que o emitiu (a autoridade de

certificação). Este envio é realizado de forma segura através de um canal TLS, usando o certificado do Painel de Controlo e o certificado do Nó de Simulação que lançou este agente. Como o certificado a enviar é publico, qualquer outro agente poderia ver o seu conteúdo, pelo que esta medida tem como principal objetivo apenas assegurar integridade.

A arquitetura descrita tem como base a confiança no Painel de Controlo e nos Nós de Simulação do sistema. Nesse sentido poderíamos fazer a geração das chaves de cada agente diretamente no Painel de Controlo ou no Nó de Simulação, reduzindo a dificuldade de enviar a chave pública para ser gerado certificado na CA do sistema. No entanto, um dos pressupostos da criptografia de chave pública é que apenas cada entidade (cada agente neste caso) deve conhecer a sua chave privada. Assim, surge a necessidade de um mecanismo mais complexo que permite a geração de certificados para os agentes, depois de eles fazerem a geração do seu par de chaves, surgindo a arquitetura anterior.

No entanto, com esta arquitetura surge um outro problema: como garantir que um agente pede a certificação usando o seu identificador e não outro (possivelmente de outro agente)? Para resolver esta vulnerabilidade, em cada Nó de Simulação obrigamos a que o identificador usado no pedido de certificação seja aquele que o agente originalmente tinha, isto é, aquele que o Painel de Controlo lhe atribuiu. Para isso, cada agente lançado por um Nó de Simulação recebe um *cryptographic nonce* como parâmetro na sua criação. Um *cryptographic nonce* consiste numa string gerada com recurso a um gerador de números aleatórios criptograficamente seguro, e que é usada apenas uma vez. No mecanismo desenvolvido, o Nó de Simulação guarda uma associação entre esta string e o identificador do agente lançado. Quando é feito um pedido de certificação, o agente que o faz tem de enviar esse *cryptographic nonce* como prova da sua identidade. Se ao pedido de certificação que o agente pretende enviar corresponder o identificador associado àquele *cryptographic nonce*, então o pedido é realizado com sucesso. Caso contrário, o Nó de Simulação descarta o que seria um pedido malicioso.

5.2.2 Comunicação Segura com o DF e o AMS

O AgentService fornece suporte para o uso do DF e do AMS segundo as normas FIPA. No entanto, a comunicação com estes componentes é realizada sem qualquer tipo de segurança. Desta forma, seria possível registar ou cancelar registos de serviços em nome de outros agentes. Para além disso, seria possível alterar os resultados de um pesquisa nas páginas amarelas ou brancas.

A solução desenvolvida consiste num *wrapper* que permite a realização destas operações de forma segura, usando o RabbitMQ. Para cada um dos serviços, DF e AMS, o *wrapper* desenvolvido é constituído por dois principais componentes: o agente responsável por fornecer a funcionalidade e o cliente que realiza pedidos para a realização das operações. O agente consiste num típico agente da plataforma que vai receber e retornar mensagens no formato FIPA ACL. Neste caso o agente funciona como um *wrapper* que apenas troca as mensagens de forma segura e faz as verificações de identidade necessárias, chamando depois o código do AgentService para realizar a operação, retornando o resultado. Este agente, para a comunicação segura com o RabbitMQ,

necessita de ter um certificado que permite assegurar a sua identidade. No contexto da plataforma do LIACC, este agente utilizará um certificado estático emitido pela CA do sistema. Cada agente da plataforma consiste num cliente dos serviços que o AMS e o DF fornecem, fazendo pedidos, sob a forma de mensagens FIPA ACL, aos agentes AMS e DF.

As operações fornecidas pelo DF permitem registar, cancelar, modificar e pesquisar por serviços nas páginas amarelas. Estas operações, definidas de acordo com a especificação FIPA, são agora realizadas de forma segura, garantindo integridade e confidencialidade em todas elas. Para todas estas operações, exceto pesquisas no registo, é também verificada a identidade do agente que a requisita, para que apenas o próprio agente possa realizar as operações que o afetem diretamente.

As operações fornecidas pelo AMS permitem registar, cancelar, modificar e pesquisar por agentes em específico ou em geral nas páginas brancas da plataforma. Tal como para o DF, estas operações podem ser realizadas garantindo integridade e confidencialidade em todas elas, e autenticidade nas operações que afetam diretamente a informação de um agente.

5.2.3 Serviço de Certificação do Sistema

O serviço de certificação do sistema, localizado no Painel de Controlo, permite a criação de certificados válidos para os agentes da plataforma. Este serviço recebe pedidos sob a forma de um objeto a que chamamos "AgentCertificationRequest". Este objeto é enviado usando o RabbitMQ e é constituído pela chave pública do agente que se pretende certificar, pelo seu identificador único e pelo endereço IP da máquina em que está a correr. A chave pública recebida precisa de ser assinada com a chave privada da autoridade de certificação, dando origem ao certificado. O identificador único permite que o certificado seja emitido em nome daquele agente. O endereço IP deve corresponder ao endereço a partir do qual o agente se liga ao *broker* do RabbitMQ. Desta forma, um agente que migre para uma outra plataforma vai necessitar de ter o seu certificado regenerado para o novo endereço IP.

Como o serviço de certificação irá receber, possivelmente, muito pedidos num espaço de tempo reduzido, este deve ser criado com vários *threads* consumidores. Cada *thread* vai ler mensagens da mesma fila do RabbitMQ, o que permite a distribuição uniforme dos pedidos pelos diversos consumidores/*threads*. Depois irá processar o pedido e responder. Neste momento, a plataforma do LIACC está a usar um número de *threads* igual ao número de núcleos da máquina em que está a correr o Painel de Controlo.

Com o objetivo de gerar certificados para os agentes consideramos duas possíveis estratégias:

- Utilização da ferramenta OpenSSL para gerar os certificados através de comandos da *shell*. Esta abordagem tem como principal vantagem o uso de uma ferramenta que segue todas as normas associadas ao uso de certificados, sendo adotada nos tutoriais do RabbitMQ e portanto completamente compatível com o mesmo [Pivotal RabbitMQ, 2020c]. Por outro lado, para a geração de certificados programaticamente e possivelmente em número elevado (igual ao número de agentes da simulação), os consecutivos acessos a disco para ler e escrever os resultados das operações tornam-se bastante custosos em termos de tempo. Esta foi

a primeira abordagem que adotamos, com o objetivo de verificar que todo o sistema estava a funcionar sem erros relacionados com o uso de certificados.

- Utilização de código nativo C# para o efeito. Esta abordagem tem como principal vantagem a criação de código mais limpo e integrado, sem necessidade de chamadas ao sistema. Para além disso, o facto de não ler e escrever do disco entre operações torna o processo de certificação bastante mais rápido e eficiente. No entanto, o uso destes mecanismo ainda apresenta muito pouca documentação quando associado ao uso de criptografia de curva elíptica, o que torna o processo bastante mais complexo de desenvolver. Apesar disso, com a ajuda de um programador que trabalha na segurança da .NET Framework, conseguimos colocar o sistema a funcionar e esta acabou por ser a abordagem adotada [Barton, 2020].

5.2.4 Registo de Agentes no RabbitMQ

A arquitetura de certificação de agentes permite-nos ter um certificado para cada agente, e assim garantir confidencialidade e integridade na troca de mensagens. No entanto, em termos de autenticidade apenas temos a garantia que o *broker* do RabbitMQ confia em quem enviou a mensagem (porque tem um certificado em que ele confia). Como vimos, o RabbitMQ apresenta uma arquitetura com o uso de *broker* pelo que não temos acesso à identidade do emissor num mensagem que chega ao recetor. O RabbitMQ apresenta, no entanto, uma propriedade que permite fazê-lo. Esta propriedade, chamada user-id, pode ser enviada juntamente com a mensagem, transportando a identidade do agente emissor [Pivotal RabbitMQ, 2020d]. O valor desta propriedade é lá colocado pelo emissor, mas a mensagem apenas é enviada se esse valor corresponder ao nome do utilizador autenticado no broker (segundo a documentação é o próprio emissor que o faz para que ele possa escolher ficar anónimo). Assim, para o uso desta propriedade todos os agentes do sistema devem estar registados no RabbitMQ.

O RabbitMQ permite o registo e gestão de utilizadores através de um *plugin* chamado rabbitmq_management que precisa de ser ativo para ser utilizado [Pivotal RabbitMQ, 2020b]. O rabbitmq_management permite o registo de utilizadores usando o formato padrão com nome de utilizador e palavra passe. Permite também fazer este registo usando apenas o nome de utilizador e sem palavra passe. Neste ultimo, uma autenticação posterior precisa de utilizar um certificado em que o *broker* confie e cujo Distinguished Name (DN) seja igual ao nome de utilizador usado no registo. O Distinguished Name é um identificador único para a identidade do certificado. Este identificador é constituído por outros campos de um certificado como o CommonName (CN), UserId (UID) e outros [Zeilenga, 2006].

O processo de registo dos agentes tem dois principais desafios: todos os agentes registados precisam de ter um certificado cujo DN seja igual ao nome de utilizador com que foram registados; e comunicação segura com o *broker* do RabbitMQ para o registo dos agentes, de forma a evitar o acesso indesejado às credenciais de acesso. O primeiro pode ser resolvido usando a arquitetura de certificação de agentes já apresentada na Fig. 5.1. O valor do DN a usar segue o formato "UID=<IdentificadorAgente>,CN=<EndereçoIpAgente>"e este deve ser o nome de utilizador de

cada agente. O *broker* do RabbitMQ verifica se o endereço no certificado apresentado por um cliente (campo CN) corresponde ao endereço do qual está a receber a ligação. Assim, é necessário o uso de CN igual ao endereço IP do Nó de Simulação onde corre o agente. O uso do UID igual ao identificador do agente serve apenas para que um agente recetor tenha acesso ao identificador do agente (AID) que lhe enviou a mensagem (através da propriedade user-id do RabbitMQ).

O registo de um agente pode ser feito, utilizando o *plugin*, de duas formas:

- Execução de comandos na máquina em que está o *broker*. Esta abordagem é, no entanto, bastante lenta e limitada.

- Usando a API HTTP/HTTPS através da interface gráfica num *browser* ou isoladamente. O uso da API torna as operações bastante rápidas e permite o acesso a um grande número de configurações. O uso de HTTPS apenas requer que o *broker* tenha um certificado em que os clientes confiam, que vai ser usado para tornar a ligação segura (usando TLS).

Na nossa implementação usamos a API HTTPS que permite a realização das operações de forma rápida e segura. O certificado usado pelo *broker* é o mesmo já usado na troca de mensagens de forma segura. No entanto, ambos os mecanismos são independentes e poderiam ser usados certificados diferentes. O registo de um agente no *broker* do RabbitMQ é realizado pelo Painel de Controlo antes de enviar cada agente para lançamento num dado Nó de Simulação. O Painel de Controlo usa as suas credenciais de acesso ao *broker*, que apenas ele deve conhecer, para registar o novo agente dando-lhe as permissões necessárias.

5.2.5 Permissões de Acesso ao RabbitMQ

O RabbitMQ permite a definição de permissões para o acesso a filas e *exchanges*. O uso de permissões permite garantir autorização no acesso às mensagens que pertencem a um agente ou a um grupo de agentes. O RabbitMQ distingue as permissões em configuração, leitura e escrita [Pivotal RabbitMQ, 2020a]. As permissões de configuração estão relacionadas com a criação de filas e *exchanges* no *broker*. As permissões de leitura estão relacionadas com a possibilidade de leitura de uma fila e respetiva associação de uma fila a um *exchange*. As permissões de escrita estão relacionadas com a possibilidade de escrita num *exchange* e a respetiva associação de uma fila a um *exchange*.

No caso da plataforma do LIACC, as permissões de configuração devem assegurar que todos os agentes devem conseguir receber mensagens individuais (fila "Agent-<IdAgente>-MessageQueue") e de um tópico subscrito (fila "Agent-<IdAgente>-<Topico>-MessageQueue") usando o seu respetivo identificador único e um qualquer tópico. Os agentes veículo, pertencentes a equipas, devem também conseguir criar filas para receberem mensagens da sua equipa (fila "Agent-<IdAgente>-Team-<IdEquipa>-MessageQueue") e do seu tipo de veículo (fila Agent-<IdAgente>-<TipoVeículo>-MessageQueue) usando os respetivos identificadores do agente, da equipa e do tipo de veículo. Da mesma forma, todos devem conseguir criar *exchanges* onde publicam mensagens para a sua equipa

(fila "BroadcastExchange-Team-<IdEquipa>"), para o seu tipo de veículo (fila "BroadcastExchange-VehicleType-<TipoVeículo>") e para um qualquer tópico (fila "BroadcastExchange-RoutingPattern-<Topico>") que decidam criar.

As permissões para leitura seguem uma estrutura igual às permissões de configuração, restringindo a leitura de mensagens ao que está associado ao agente (comunicação direccionada a ele, à sua equipa, ao seu tipo de veículo ou a um qualquer tópico).

As permissões de escrita são menos restritivas e permitem que um agente envie mensagens para qualquer outro agente, equipa, tipo de veículo e tópico. O agente que recebe a mensagem é depois responsável por, tendo em conta a identidade do emissor (propriedade user-id), considerar a mensagem confiável ou não.

5.2.6 Níveis de Segurança

Desde o início sabíamos que o uso de segurança na comunicação implicaria um decréscimo de desempenho significativo, tal como os testes de desempenho aos *middleware* e MAS mostraram (secção 4.3). Por isso decidimos colocar suporte para utilizar a plataforma com três níveis de segurança distintos: autenticação e TLS (nível padrão), apenas TLS e sem segurança. Na realidade, tentamos também o uso de apenas autenticação, mas o *plugin* que permite autenticação usando certificados X509 apenas funciona com o TLS ativo [Pédron et al., 2020]. A configuração de que nível de segurança usar poderia ser feita por tipo de agente, por subscrição e por produtor de forma independente (por exemplo, um agente pode produzir mensagens usando autenticação e TLS, subscrever mensagens individuais apenas com TLS e subscrever mensagens de equipa de forma insegura). No entanto, por facilidade de uso, a interface do Painel de Controlo apenas faz a distinção por tipo de agente, em que por exemplo um agente ATC pode estar a utilizar um nível de segurança diferente de um agente veículo.

Para além da melhoria de desempenho que o não uso de segurança permite obter durante a realização de uma simulação, com as mensagens a serem trocadas mais rapidamente, também se conseguem obter melhorias nos tempos de lançamento da simulação. Quando está a ser utilizada a versão insegura, todas as operações associadas à segurança (como o registo dos agentes no *broker* e a certificação dos agentes) são ignoradas. Esta será um boa opção, por exemplo, para uso durante o desenvolvimento, em que a plataforma precisa de ser lançada possivelmente muitas vezes.

A Tabela 5.2 mostra a evolução dos tempos obtidos para as operações de registo no *broker* do RabbitMQ e para a certificação dos agentes, com o aumento do número de agentes. O processo de registo de um agente no *broker* é realizado, pelo Painel de Controlo, antes do envio de um agente para o respetivo Nó de Simulação, sendo este processo feito de forma iterativa e sequencial. Considerando os 3 níveis de segurança, este processo é realizado apenas quando se usa autenticação juntamente com TLS. O processo de certificação dos agentes é realizado dentro de cada agente e, por isso, de forma paralela entre eles. Tal como descrito na secção 5.2.3, a autoridade de certificação utiliza vários *threads* responsáveis pela certificação, de forma a tirar partido deste paralelismo. Os resultados apresentados na tabela correspondem a testes realizados utilizando 4 *threads* consumidores na CA do Painel de Controlo, permitindo assim o processamento

de 4 pedidos em paralelo. Este processo é realizado quando se usa autenticação juntamente com TLS, bem como quando se usa apenas TLS.

Tabela 5.2: Testes de Desempenho para o Registo e Certificação

	10 agentes	**50 agentes**	**100 agentes**	**500 agentes**	**1000 agentes**
Tempo Registo (s)	0.33	1.37	2.72	13.24	28.23
Tempo Certificação (s)	0.31	1.15	2.36	11.77	26.24
Total (s)	0.63	2.51	5.08	25.01	54.47

Os valores mostram um aumento dos tempos aproximadamente linear com o aumento do número de agentes. Os valores dos dois processos (registo e certificação) são bastante próximos, apesar da operação de certificação utilizar computação bastante mais intensiva. Isto pode ser explicado pelo uso de paralelismo no processo de certificação, em contraste com o processo sequencial usado no registo dos agentes.

5.3 Distribuição dos Agentes Externos

Na plataforma do LIACC a arquitetura criada para a distribuição de agentes, que pode ser vista na Fig 4.15, usa o Painel de Controlo como o nó responsável pelo lançamento dos agentes em execução. O Painel de Controlo recebe ordens do seu operador, através da interface gráfica (GUI), para o lançamento de um conjunto de agentes. Nessa altura, o Painel de Controlo verifica que Nós de Simulação estão ativos no sentido de lhes enviar os novos agentes para correrem lá. O lançamento de Nós de Simulação pode também ser feito remotamente, usando a GUI do Painel de Controlo, como descrito na secção 5.3.1. O envio de cada agente é feito sob a forma de uma nova tarefa, um objeto base do tipo "AgentSimulationTask", que contém o identificador do agente e os parâmetros para a sua criação. Este envio é feito para o Nó de Simulação que naquele momento tem menos agentes a correrem sob a sua responsabilidade. Consideramos também a possibilidade de diferentes Nós de Simulação terem capacidades de processamento diferentes, mas essa métrica não está a ser utilizada neste momento. A troca destas mensagens/objetos entre o Painel de Controlo e os Nós de Simulação é feita usando o RabbitMQ de forma segura, cada um com o seu certificado como descrito na secção 5.2.

Em cada Nó de Simulação temos um serviço que está sempre à espera de tarefas que vêm sob a forma de mensagens, uma "AgentSimulationTask". Cada tipo de agente é depois um objeto derivado com o comportamento de lançamento especifico. Na plataforma do LIACC temos: VehicleAgentSimulationTask para os agentes veículo, ATCAgentSimulationTask para os agentes ATC e DisturbancesManagerAgentSimulationTask para o *Disturbances Manager*.

5.3.1 Lançamento Remoto dos Nós de Simulação

A GUI do Painel de Controlo permite a criação e o lançamento remoto dos Nós de Simulação. Esse lançamento é feito com recurso ao protocolo SSH[6] (*Secure Shell*) que permite a execução de comandos em máquinas remotas de forma segura, usando o conceito de encriptação de chave pública. Este sistema envolve duas entidades: o cliente que quer executar comandos numa outra máquina; e o servidor onde vão ser executados os comandos. Ambas as entidades têm um par de chaves (chave pública e chave privada). De forma a garantir ao cliente que este se está a ligar ao servidor correto, o cliente deve possuir a *fingerprint* (impressão digital) do servidor. A *fingerprint* corresponde a um conjunto de bytes que representam o *hashing* criptográfico da chave pública do servidor. Quando o cliente se liga ao servidor, este verifica se o *hashing* criptográfico da chave pública que o servidor está a usar é igual ao que o cliente já tinha guardado e sabe ser confiável. Do lado do servidor, este também precisa de verificar que quem se tenta ligar (e executar comandos) é de confiança, pelo que a chave pública do cliente deve estar já presente no servidor como confiável. Assim, ambas as entidades se autenticam e podem comunicar de forma segura.

Quando comparado com outros métodos de execução remota de comandos, nomeadamente o WinRM[7] e o PsExec[8], a configuração do protocolo SSH é bastante mais simples para o uso de comunicação segura, como referido em [Wrock, 2016] e em [Korneck, 2017].

No contexto da plataforma do LIACC as informações que o Painel de Controlo (cliente) precisa de ter sobre cada Nó de Simulação (servidor) são o seu identificador único, o seu endereço IP e a sua *fingerprint*. Cada Nó de Simulação deve ter a chave pública do Painel de Controlo marcada como confiável. Assim, o Painel de Controlo consegue ligar-se a cada Nó de Simulação usando o seu endereço IP e o nome de utilizador igual ao seu identificador único. Depois é só executar o comando que lança o Nó de Simulação.

Na plataforma do LIACC, antes do arranque de uma simulação é necessário definir o cenário, as equipas e as missões. Isto pode ser feito usando um ficheiro XML, usando a GUI do Painel de Controlo ou ambos (as alterações feitas usando a GUI são depois guardadas no ficheiro respetivo). Estes dados são depois utilizados pelos agentes durante a simulação. Como os agentes vão correr num conjunto de máquinas bem definido (os Nós de Simulação), estes ficheiros são enviados para cada nó simulação aquando do seu lançamento, em vez de para cada agente, permitindo que os agentes tenham o ficheiro disponível para usar sempre que precisarem. Como os ficheiros podem ser bastante grandes, reduzimos o tempo de lançamento de uma simulação. Para o envio dos ficheiros, aquando do lançamento de um Nó de Simulação, usamos o protocolo SFTP[9], que permite a troca de ficheiros de forma segura baseada em SSH. Os mecanismos de segurança usados são por isso os já descritos anteriormente para o protocolo SSH.

[6] Mais informação disponível em https://www.ssh.com/ssh/protocol/

[7] Mais informação disponível em https://docs.microsoft.com/en-us/windows/win32/winrm/portal

[8] Mais informação disponível em https://docs.microsoft.com/en-us/sysinternals/downloads/psexec

[9] Mais informação disponível em https://www.ssh.com/ssh/sftp/#sftp-protocol

5.3.2 Testes de Distribuição dos Componentes da Plataforma

A plataforma do LIACC tem agora para a distribuição dos agentes por diversos computadores, os Nós de Simulação. Assim, tendo como objetivo uma maior capacidade de simulação, os diversos componentes da plataforma devem poder ser distribuídos por várias máquinas. Desta forma, foram realizados alguns testes para verificar se era possível colocar o *broker* do RabbitMQ, o FSX e o Painel de Controlo todos em máquinas diferentes e a comunicarem pela rede.

O *broker* do RabbitMQ não apresentou qualquer problema no estabelecimento das ligações e execução das operações necessárias. Com o uso de TLS para a comunicação com o *broker* (quer para o envio e receção de mensagens, quer para as operações de gestão de utilizadores) todo o mecanismo é seguro para ser usado na rede.

O FSX, por outro lado, apresentou alguns problemas. Na plataforma do LIACC, o FSX funciona como um componente externo que faz a simulação dos parâmetros associados aos agentes. Cada agente envia e recebe informações do FSX utilizando uma API desenhada para o efeito, o SimConnect fornecido pelo SDK (*Software Development Kit*) do FSX [Microsoft, 2008]. Um agente da plataforma utiliza o SimConnect para se ligar ao FSX, por isso é necessário que em cada Nó de Simulação (onde correm os agentes) esteja instalado o SDK do FSX. Nesse sentido e de acordo com [FSDeveloper, 2020] instalamos todos os pré-requisitos do SDK e o próprio SDK num Nó de Simulação. No entanto, não conseguimos correr os agentes da plataforma devido à falta de dependências que apenas o próprio FSX parece instalar, pelo que tivemos de instalar o FSX em cada Nó de Simulação. De notar que até agora a plataforma do LIACC sempre correu os agentes num computador em que para além do SDK também o FSX estava instalado, e este problema nunca ocorreu. No entanto, o FSX é um jogo que ocupa aproximadamente 17GB, pelo que tê-lo instalado em cada Nó de Simulação é relativamente pesado para os objetivos do LIACC. Como apenas precisávamos das dependências que o FSX instala, decidimos então remover os ficheiros do jogo e assim eliminar o problema do espaço que o jogo ocupa (desinstalando o jogo, as dependências necessárias também eram desinstaladas e o problema voltava). Seria importante referir que em paralelo com esta dissertação, está a ser realizada uma outra dissertação para permitir o uso da platafroma do LIACC com várias instâncias do FSX em múltiplas máquinas [Esteves, 2020]. Assim, surge a oportunidade de uma mesma máquina correr ambos um Nó de Simulação e uma instância do FSX distribuído, minimizando este problema. Para as máquinas que apenas correm um Nó de Simulação, a solução final consiste na instalação do FSX e do SDK nessas máquinas e posterior remoção dos ficheiros do FSX. Em termos de comunicação, o SimConnect não apresenta qualquer mecanismo de segurança, nem encontramos implementações que o permitam, pelo que as comunicações entre os agentes e o FSX podem ser adulteradas quando usadas na rede.

O Painel de Controlo funciona como um típico agente da plataforma do LIACC, pelo que usa o RabbitMQ para a comunicação segura com os restantes agentes. Para além disso, o Painel de Controlo verifica as ligações aos outros componentes como o FSX e os Nós de Simulação, antes do arranque de uma simulação. Nos testes que realizámos, com a distribuição dos componentes por diversas máquinas, o Painel de Controlo manteve o funcionamento esperado.

Capítulo 6

Avaliação da Solução

De forma a avaliar o impacto da solução desenvolvida na plataforma do LIACC foram realizados testes de desempenho à plataforma, com o uso do seu *middleware* original (o AgentService) e com o uso do novo *middleware* (o RabbitMQ). No caso do RabbitMQ, os testes foram realizados sem e com o uso de segurança para dar um panorama global e comparativo das duas soluções.

Os testes foram realizados para os mesmos cenários dos testes de desempenho já realizados anteriormente e descritos na secção 4.3, isto é, comunicação um-para-um, um-para-muitos e muitos-para-um. Assim conseguimos comparar também o custo de integração do RabbitMQ na plataforma. Este custo está associado ao uso de mensagens com um tamanho superior, bem como ao tempo de serialização da mensagem (este tempo foi considerado para os testes pois o *middleware* do AgentService também o faz internamente e assim os resultados são mais justos e facilmente comparáveis).

Para além disso, foram realizados testes comparativos para um caso especifico da comunicação da plataforma do LIACC, entre o *Disturbances Manager* e os agentes veículo. Esta comunicação já tinha sido avaliada em [Almeida, 2017], tendo sido considerada um *bootleneck* para a plataforma. Com a repetição destes testes, esperamos perceber a evolução no desempenho que o uso do RabbitMQ permite nesse cenário.

6.1 Comunicação Um-para-Um

Neste cenário, os resultados do AgentService são bastante fracos, apresentando latências extremamente elevadas, como pode ser visto na Fig. 6.1, e uma taxa de transferência de mensagens bastante baixa, como pode ser visto na Fig. 6.2. Assim, o RabbitMQ apresenta resultados muito melhores sem o uso de segurança e até com o uso de segurança. Mesmo quando comparado com os resultados das plataformas multi-agente já testadas, o AgentService apresenta um desempenho bastante inferior. No entanto, os testes ao AgentService foram realizados com o uso de agentes externos, enquanto que nas outras plataformas foram usados agentes *in-platform*.

No que diz respeito apenas aos resultados do RabbitMQ, os valores apresentados sofreram algumas alterações quando comparados com os resultados originais do *middleware* isolado. O

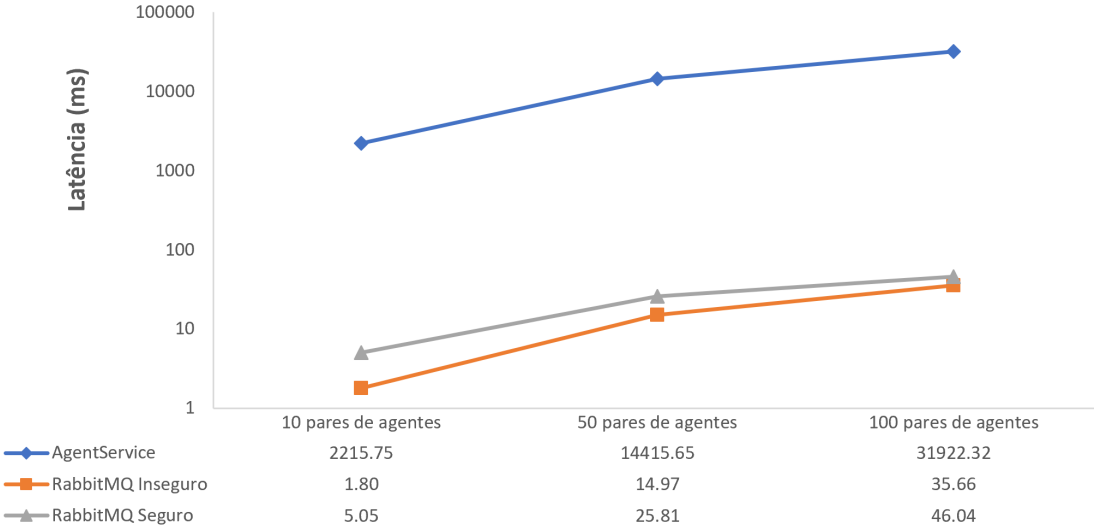


Figura 6.1: Latência na Troca de Mensagens - Um-para-Um - Antes e Depois

decréscimo de desempenho a que assistimos pode ser justificado com a introdução do processo de serialização da mensagem, bem como pelo aumento do tamanho das mensagens trocadas. Em termos de latência assistimos apenas a um pequeno aumento. Quanto à taxa de mensagens enviadas, registamos um impacto superior, principalmente nos cenários com mais pares de agentes. Os resultados são, ainda assim, bastante positivos, na medida em que se mantém superiores à maioria das outras plataformas estudadas, principalmente com o uso de segurança.

6.2 Comunicação Um-para-Muitos

Neste cenário, os resultados mostram novamente o AgentService com um desempenho muito inferior ao do RabbitMQ seguro e inseguro. Os valores das latências apresentadas podem ser vistos na Fig. 6.3 e as taxas de transferência de mensagens na Fig. 6.4.

No que diz respeito apenas aos resultados do RabbitMQ, voltamos a assistir a um decréscimo do desempenho quer na latência, quer na taxa de mensagens enviadas. Em termos de latência o impacto foi bastante mais acentuado nos cenários com apenas 10 produtores, com e sem o uso de segurança, devido aos valores bastante baixos que o RabbitMQ apresentou de forma isolada. No entanto, estes valores continuam a ser melhores que todos os outros MAS e *middleware*. Para os cenários com 50 produtores os resultados são mais próximos dos originais. Em relação à taxa de mensagens enviadas, voltamos a assistir a uma diminuição acentuada, mas mantendo valores bastante elevados e por isso bastante bons.

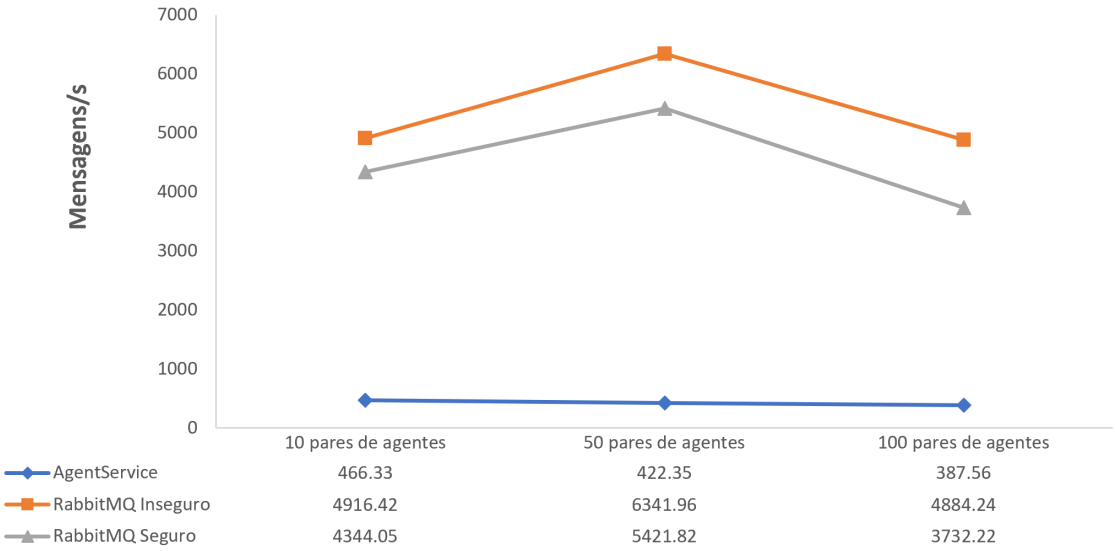


Figura 6.2: Taxa de Transferência de Mensagens - Um-para-Um - Antes e Depois

6.3 Comunicação Muitos-para-Um

Para a comunicação muitos-para-um, os resultados mostram novamente o AgentService com latências muito superiores às do RabbitMQ seguro e inseguro, como pode ser visto na Fig. 6.5. Em termos da taxa de transferência de mensagens, o AgentService apresenta valores muito baixos e muito inferiores aos do RabbitMQ, como pode ser visto na Fig. 6.6.

No que diz respeito apenas aos resultados do RabbitMQ, assistimos a um pequeno aumento na latência apresentada, quando comparando com os resultados originais do *middleware* isolado. Este aumento foi mais acentuado na versão com o uso de segurança, mas com o RabbitMQ a manter-se melhor que todos os MAS e *middleware* testados. Quanto à taxa de transferência de mensagens, o impacto foi mais acentuado sem o uso de segurança e nas versões com maior número de produtores. Os resultados continuam, no entanto, bastante elevados e por isso bastante bons.

6.4 Comunicação *Disturbances Manager* - Agentes Veículo

Devido à grande necessidade de comunicação entre o DM e os agentes veículo, para a comunicação das leituras dos sensores associados aos distúrbios, decidimos perceber a evolução de desempenho que o DM conseguiu obter, usando agora o novo *middleware*, o RabbitMQ. Este tipo de comunicação já tinha sido testada em [Almeida, 2017], tendo mostrado que o AgentService era um *bottleneck* para o desempenho do DM. Assim, decidimos repetir esse conjunto de testes, que consistem em dois tipos: testes de stress e testes de resistência.

Os testes de stress permitem encontrar um valor estável da taxa de mensagens enviadas por agente para um dado número de agentes. Este teste corre durante um minuto, em que temos um determinado número de mensagens que esperamos enviar (taxa de mensagens por segundo * 60).

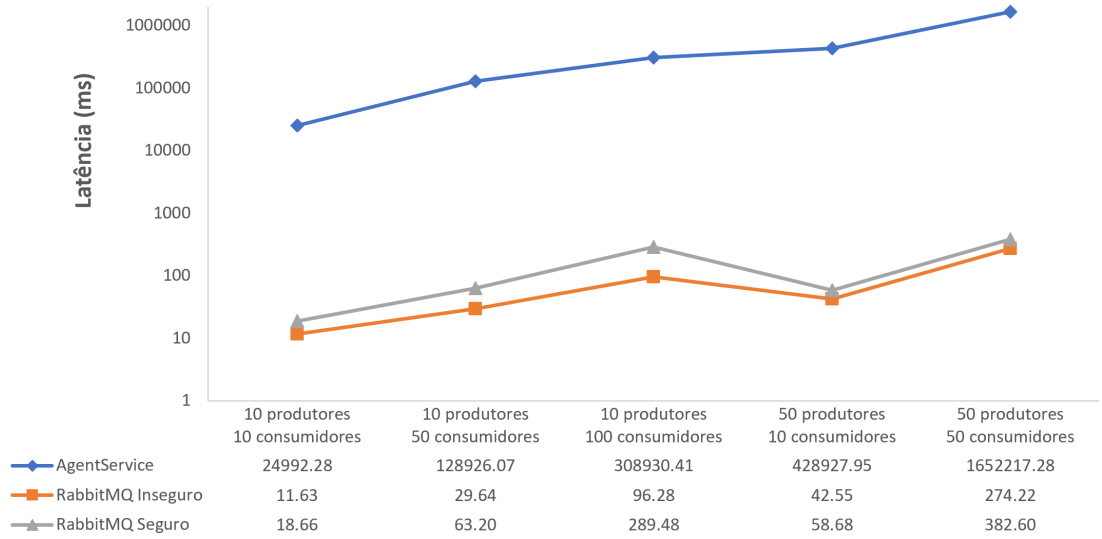


Figura 6.3: Latência na Troca de Mensagens - Um-para-Muitos - Antes e Depois

De forma a encontrar a taxa mais elevada que o *middleware* permite, são testadas várias taxas até atingir uma em que o valor do número de mensagens que esperamos enviar não consiga ser atingido durante os 60 segundos do teste. Entre cada iteração de envio de uma nova leitura para todos os agentes, e de forma a manter a taxa desejada, surge a necessidade de esperar um valor de tempo estabelecido. Este valor corresponde ao inverso da taxa de mensagens que se pretende enviar, adaptado para considerar o tempo de envio das mensagens (que é superior a 0). Este valor é adaptado a cada segundo, de acordo com o valor atual de mensagens enviadas e o valor que se esperaria já ter sido enviado naquele momento. O valor será adaptado de um fator de 0.9 quando o número de mensagens enviadas é superior ao esperado e de um fator de 1.1 quando o número de mensagens enviadas é inferior ao esperado. Os resultados apresentados abaixo mostram apenas a taxa encontrada como solução e a taxa seguinte; no entanto, tiveram de ser testados outros valores intermédios até se atingir a taxa solução.

Os testes de resistência seguem o mesmo formato, mas testam o valor base da taxa de mensagens enviadas por agente durante 30 minutos. Com isto espera-se confirmar se o *middleware* conseguiria manter aquela taxa durante bastante tempo. Os resultados apresentados abaixo mostram o resultado da tentativa de obter a mesma taxa encontrada no teste de stress e o resultado da taxa que conseguiu manter-se durante os 30 minutos (taxa solução). Quando estas duas taxas coincidem apenas mostramos uma, indicando que a taxa obtida no teste de stress se conseguiu manter durante muito tempo.

A Tabela 6.1 mostra os valores obtidos para a comunicação entre o DM e os agentes veículo usando o AgentService. Um teste com a mesma configuração já tinha sido feito em [Almeida, 2017], no entanto, para um número diferente de agentes e usando um CPU diferente. Assim, para a comparação com os resultados que apresentamos a seguir, decidimos repetir os testes realizados. Conseguimos obter, no teste de stress, uma taxa de 12 mensagens por segundo por agente, com

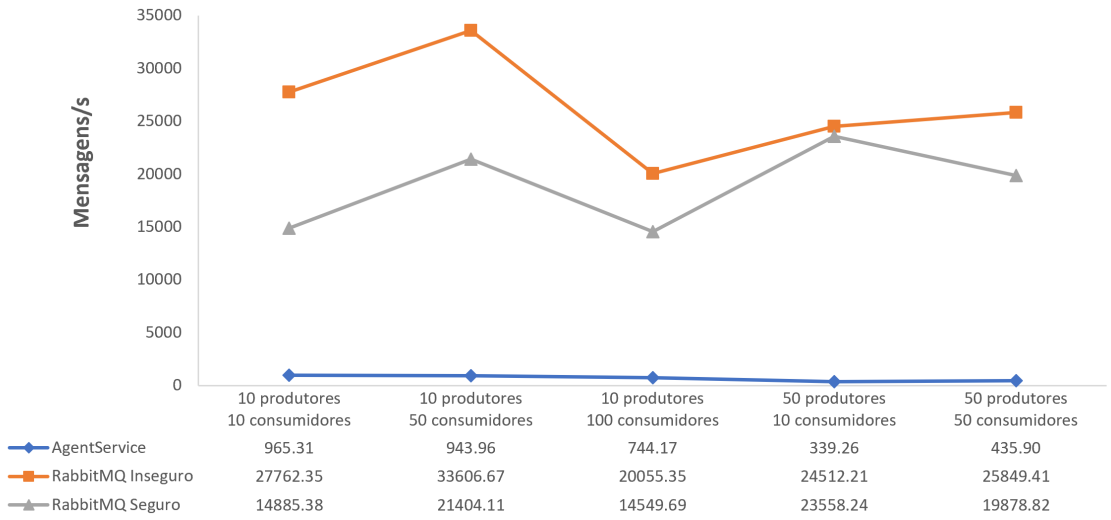


Figura 6.4: Taxa de Transferência de Mensagens - Um-para-Muitos - Antes e Depois

o DM a enviar mensagens para 20 agentes veículo em simultâneo. Para 50 agentes veículo em simultâneo, este valor desce para apenas 2 mensagens por segundo por agente. Ambos os valores foram confirmados com o teste de resistência.

Tabela 6.1: Testes de Desempenho da Comunicação de Distúrbios usando o AgentService

Teste	**Nº Agentes**	**Mensagens/s /agente**	**Tempo (segundos)**	**Mensagens/s/agente Efetivas**	**Mensagens/s Efetivas**
Stress	20	12	59.69	12.06	241.23
	20	13	61.39	12.71	254.12
	50	2	59.85	2.01	100.26
	50	3	67.38	2.67	133.57
Resistência	20	12	1789.05	12.07	241.47
	50	2	1773.40	2.03	101.50

Os mesmos testes foram realizados usando o RabbitMQ com o DM a usar apenas um *thread*, tal como com o AgentService, e de acordo com a arquitetura atual da plataforma do LIACC. Neste momento, todos os distúrbios da plataforma estão a ser controlados pelo DM, que internamente usa apenas um *thread* para o cálculo e para o envio das leituras dos sensores para todos os agentes veículo. Os testes foram agora realizados sem e com o uso de segurança.

Sem o uso de segurança, os resultados da Tabela 6.2 mostram uma taxa de 115 mensagens por segundo por agente, com 20 agentes veículo em simultâneo (10 vezes melhor que o AgentService). Para 50 agentes veículo em simultâneo, este valor desce para 33 mensagens por segundo por agente (17 vezes melhor que o AgentService). Ambos os valores foram confirmados com o teste de resistência.

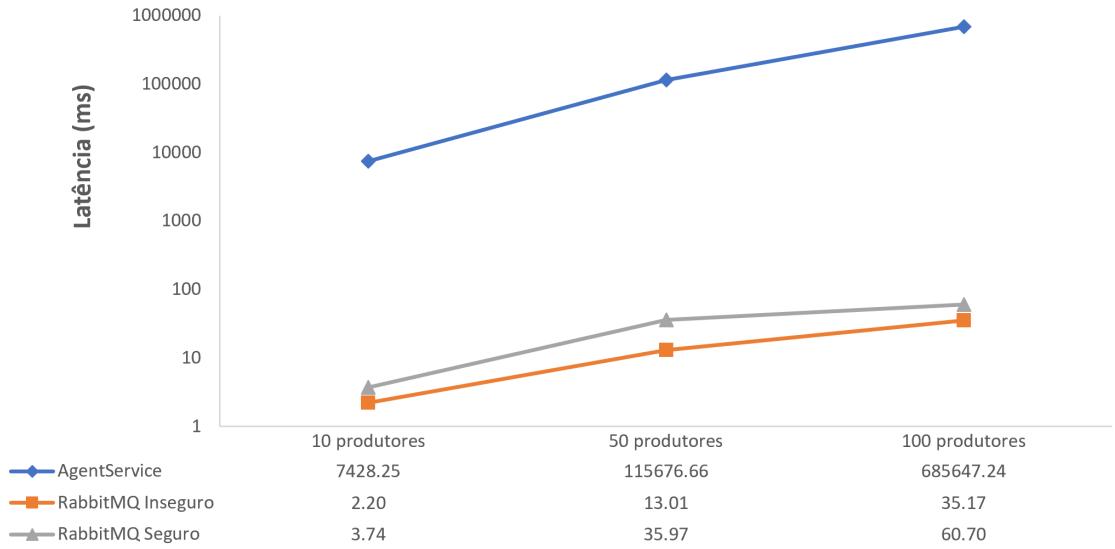


Figura 6.5: Latência na Troca de Mensagens - Muitos-para-Um - Antes e Depois

Com o uso de segurança, os resultados da Tabela 6.3 mostram que a taxa desce para 99 mensagens por segundo por agente, com 20 agentes veículo em simultâneo, registando uma descida de 13.91%. Para 50 agentes veículo em simultâneo, este valor desce para 24 mensagens por segundo por agente, registando uma descida de 27.27%. Ambos os valores foram confirmados com o teste de resistência. Os resultados obtidos foram bastante melhores que os do AgentService, mesmo com o uso de segurança (em média 9 vezes melhores).

O *Disturbances Manager* envia para cada agente as leituras correspondentes aos seus sensores, de acordo com alguns fatores, como a posição do veículo em relação a um determinado distúrbio. Este envio é realizado num ciclo para todos os agentes que devem receber informação de um determinado distúrbio. No cenário anterior, os resultados obtidos, embora bastante melhores que os resultados do AgentService, limitam bastante o número de agentes, sensores e taxas de

Tabela 6.2: Testes de Desempenho da Comunicação Insegura de Distúrbios usando o RabbitMQ com 1 *Worker*

Teste	**Nº Agentes**	**Mensagens/s /agente**	**Tempo (segundos)**	**Mensagens/s/agente Efetivas**	**Mensagens/s Efetivas**
Stress	20	115	59.98	115.05	2300.94
	20	116	60.15	115.71	2314.29
	50	33	59.93	33.04	1652.02
	50	34	60.12	33.93	1696.63
Resistência	20	115	1789.37	115.68	2313.66
	50	33	1790.44	33.18	1658.81

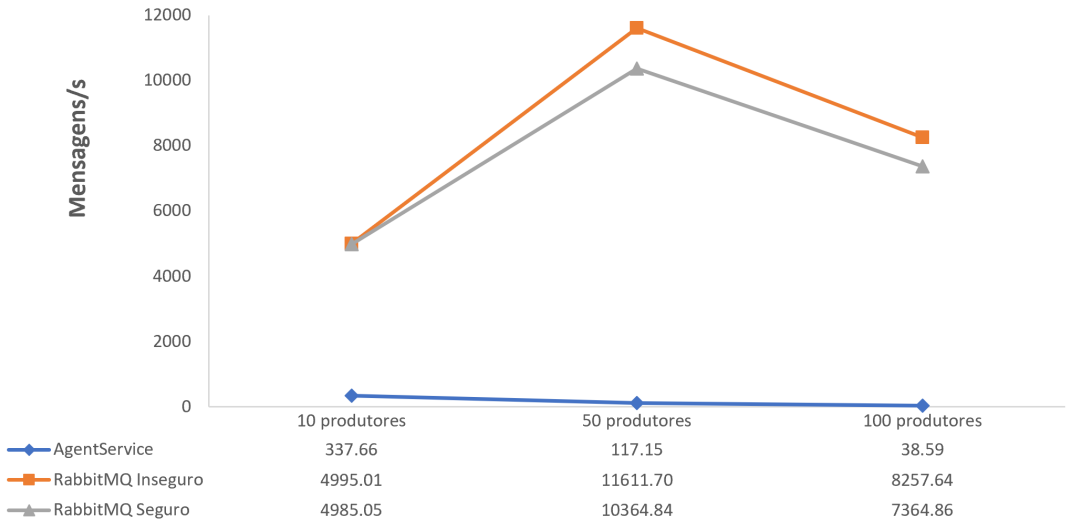


Figura 6.6: Taxa de Transferência de Mensagens - Muitos-para-Um - Antes e Depois

atualização que o DM pode usar. Isto acontece porque o envio de uma mensagem pelo DM usa apenas um *thread*, e apenas uma instância do produtor ligada ao *broker* do RabbitMQ. O uso de apenas um *thread* implica que só possa ser enviada uma mensagem da cada vez. Para além disso, o uso de apenas uma instância de produção vai limitar o envio de mensagens devido à necessidade de um processo iterativo, dado que a instância de produção só pode estar a ser usada para enviar uma mensagem de cada vez. De forma a tentar paralelizar este processo de envio das leituras dos sensores, do DM para os agentes veículo, existem três cenários possíveis: divisão das responsabilidades dentro de um DM, usando um *worker* responsável por cada distúrbio existente; manter o DM responsável por todos os distúrbios, mas criando depois um *thread* para a atualização de cada agente veículo (conjugado com uma instância do produtor de RabbitMQ por *thread*); ou a conjugação de ambos, em que podemos ter um *worker* por distúrbio, com cada um deles a ter N *threads* para atualizar N agentes veículo.

Tabela 6.3: Testes de Desempenho da Comunicação Segura de Distúrbios usando o RabbitMQ com 1 *Worker*

Teste	**Nº Agentes**	**Mensagens/s /agente**	**Tempo (segundos)**	**Mensagens/s/agente Efetivas**	**Mensagens/s Efetivas**
Stress	20	99	59.60	99.67	1993.31
	20	100	60.08	99.87	1997.49
	50	24	59.98	24.01	1200.35
	50	25	60.13	24.94	1247.23
Resistência	20	99	1774.41	100.43	2008.55
	50	24	1789.98	24.13	1206.72

Para testar o comportamento obtido com o uso de vários *workers* por distúrbio decidimos correr os mesmo testes, mas agora com o uso de 2 *workers*, de forma a mostrar a tendência de evolução seguida (o cenário anterior em que temos apenas o DM é equivalente a um cenário em que se usa 1 *worker* no DM). Este cenário seria benéfico numa situação em que temos 2 distúrbios a acontecer em simultâneo, permitindo perceber qual o valor máximo da conjugação da taxa de atualização com o número de sensores do distúrbio. A taxa de atualização, multiplicada pelo número de sensores do distúrbio, não pode ultrapassar o valor máximo de mensagens que o *worker* responsável por esse distúrbio consegue enviar. Este cenário, para além da melhoria de desempenho esperada na comunicação, permitiria distribuir a carga de processamento dos distúrbios por vários *workers*, que podiam depois ser distribuídos por várias máquinas.

Sem o uso de segurança, os resultados da Tabela 6.4 mostram uma taxa de 232 mensagens por segundo por agente, com 20 agentes veículo em simultâneo. Para 50 agentes veículo em simultâneo, obtivemos 68 mensagens por segundo por agente. Ambos os valores foram confirmados com o teste de resistência.

Tabela 6.4: Testes de Desempenho da Comunicação Insegura de Distúrbios usando o RabbitMQ com 2 *Workers*

Teste	Nº Agentes	Mensagens /agente/DM	Mensagens/s /agente	Tempo (segundos)	Mensagens/s/agente Efetivas	Mensagens/s Efetivas
Stress	20	116	232	59.87	232.49	4649.82
	20	117	234	60.33	232.73	4654.55
	50	34	68	59.87	68.14	3407.22
	50	35	70	60.13	69.85	3492.64
Resistência	20	116	232	1789.64	233.34	4666.86
	50	34	68	1790.75	68.35	3417.57

Com o uso de segurança, os resultados da Tabela 6.5 mostram que a taxa desce para 198 mensagens por segundo por agente, com 20 agentes veículo em simultâneo, registando uma descida de 14.66%. Para 50 agentes veículo em simultâneo, este valor desce para 62 mensagens por segundo por agente, registando uma descida de 8.82%. Ambos os valores foram confirmados com o teste de resistência.

Tabela 6.5: Testes de Desempenho da Comunicação Segura de Distúrbios usando o RabbitMQ com 2 *Workers*

Teste	Nº Agentes	Mensagens /agente/DM	Mensagens/s /agente	Tempo (segundos)	Mensagens/s/agente Efetivas	Mensagens/s Efetivas
Stress	20	99	198	59.46	199.80	3995.96
	20	100	200	60.08	199.73	3994.59
	50	31	62	59.99	62.01	3100.49
	50	32	64	60.18	63.81	3190.59
Resistência	20	99	198	1760.80	202.41	4048.15
	50	31	62	1791.97	62.28	3113.89

Os resultados mostram uma evolução aproximadamente linear (a taxa de mensagens duplicou com o uso de 2 *workers*), quando comparados com os resultados obtidos com o uso do DM sem *workers* (equivalente a 1 *worker*). Assim, os valores dos testes com o uso de apenas um *worker* servem de valor teórico que cada novo *worker* deverá conseguir atingir, porque o *bottleneck* está a acontecer na taxa de mensagens que uma única instância de RabbitMQ consegue enviar num determinado espaço de tempo.

Como podemos ter muitos veículos para quem o DM precisar de reportar leituras, uma outra solução é paralelizar este envio com várias instâncias do produtor de RabbitMQ, uma por cada agente veículo, na tentativa de melhorar os resultados obtidos. Este cenário é equivalente ao cenário de um-para-um testado anteriormente, em que temos um número de consumidores igual ao número de produtores, e estes comunicam aos pares. Neste caso, o DM utiliza um único *thread* para o calculo das informações de todos os distúrbios, mas depois utiliza N *threads*, cada um com uma instância de produção do RabbitMQ, para o envio das mensagens para cada um dos N agentes veículo (os consumidores).

Os testes sem o uso de segurança, cujos resultados podem ser vistos na Tabela 6.6, mostram que conseguimos obter, no teste de stress, uma taxa de 829 mensagens por segundo por agente, com 20 agentes veículo em simultâneo (7 vezes melhor que o cenário com apenas 1 produtor). Com 50 agentes veículo em simultâneo, este valor desce para 357 mensagens por segundo por agente (11 vezes melhor que o cenário com apenas 1 produtor). Neste cenário, devido à maior capacidade de envio de mensagens do DM, decidmos testar ainda o uso de 100 agentes veículos para o qual obtivemos uma taxa de 122 mensagens por segundo por agente. Todos os valores foram confirmados com o teste de resistência.

Tabela 6.6: Testes de Desempenho da Comunicação Insegura de Distúrbios usando o RabbitMQ com N Produtores

Teste	**Nº Agentes**	**Mensagens/s /agente**	**Tempo (segundos)**	**Mensagens/s/agente Efetivas**	**Mensagens/s Efetivas**
Stress	20	829	59.49	836.07	16721.40
	20	830	60.17	827.59	16551.87
	50	357	59.78	358.31	17915.39
	50	358	60.06	357.63	17881.37
	100	122	59.95	122.09	12209.28
	100	123	60.36	122.27	12227.33
Resistência	20	829	1691.38	882.24	17644.71
	50	357	1775.84	361.86	18092.84
	100	122	1785.34	123.00	12300.16

Com o uso de segurança, os resultados da Tabela 6.7 mostram que conseguimos obter, no teste de stress, uma taxa de 676 mensagens por segundo por agente, com 20 agentes veículo em simultâneo (decréscimo de 18.46% devido ao uso de segurança). Com 50 agentes veículo em

Tabela 6.7: Testes de Desempenho da Comunicação Segura de Distúrbios usando o RabbitMQ com N Produtores

Teste	Nº Agentes	Mensagens/s /agente	Tempo (segundos)	Mensagens/s/agente Efetivas	Mensagens/s Efetivas
Stress	20	676	59.98	676.25	13524.98
	20	677	60.27	673.98	13479.67
	50	275	59.14	278.99	13949.54
	50	276	60.84	272.18	13609.00
	100	94	59.81	94.30	9430.19
	100	95	60.34	94.46	9445.89
Resistência	20	676	1742.58	698.27	13965.48
	50	275	1757.78	281.61	14080.26
	100	94	1783.78	94.85	9485.48

simultâneo, obtivemos 275 mensagens por segundo por agente (decréscimo de 22.97%). Com 100 agentes veículos obtivemos 94 mensagens por segundo, resultando num decréscimo de 22.95%. Os resultados obtidos foram em média 9 vezes melhores que os resultados do cenário com apenas 1 produtor. Todos os valores foram confirmados com o teste de resistência.

A terceira opção seria o uso de um híbrido entre as duas anteriores, permitindo paralelizar ao máximo o envio de mensagens com as leituras dos sensores. Este cenário usaria um *worker* responsável por cada distúrbio, e cada um desses *workers* teria à sua responsabilidade N *threads*, e N instâncias de produção do RabbitMQ, para reportar as leituras a cada um dos N agentes veículo. Nos últimos testes, em que usamos várias instâncias do produtor de RabbitMQ com apenas um *worker* responsável pelo distúrbios (o próprio DM), observamos já a utilização máxima do CPU em que estes estavam a executar. De notar que os testes são realizado com todos os agentes, DM e *broker* do RabbitMQ na mesma máquina. Devido a este cenário, os testes com esta opção híbrida teriam o seu desempenho limitado na nossa configuração de testes, pelo que decidimos não realizar estes testes. No entanto, numa configuração com os agentes veículo e os *workers* do DM distribuídos por várias máquinas, bem como o uso do *broker* do RabbitMQ numa outra máquina ou o uso de vários *brokers* em federação, esta solução deveria conseguir atingir ainda melhor desempenho que o cenário anterior.

Capítulo 7

Conclusões e Trabalho Futuro

Esta dissertação tinha como principais objetivos assegurar comunicação rápida, eficiente (baseada em eventos) e segura na plataforma do LIACC, mantendo o suporte FIPA e a capacidade de integração de agentes externos. Esperava-se construir uma solução genérica que pudesse ser aplicada também noutras plataformas multi-agente.

Para tal, foi necessário realizar um estudo do estado da arte nas vertentes de comunicação eficiente, comunicação segura e comunicação entre agentes. Inicialmente, foram estudadas plataformas multi-agente, de forma a perceber quais as suas características, principalmente no que diz respeito à comunicação, vantagens e desvantagens. Foram escolhidas e estudadas em detalhe quatro plataformas: AgentService (usado na comunicação entre agentes na plataforma do LIACC antes desta dissertação), JADE, SPADE e JIAC. Como pretendíamos melhorar a troca de mensagens entre agentes, foram também estudados *middleware* de comunicação orientados a mensagens. Foram escolhidos e estudados em detalhe quatro *middleware*: ZeroMQ, RabbitMQ, Apache Kafka e ActiveMQ Artemis. Em relação à segurança na comunicação foram estudados os diversos mecanismos para perceber que garantias trazem e como podem funcionar em conjunto para obter um solução globalmente segura.

Foram também realizados um conjunto de testes para perceber as diferenças de desempenho entre os MAS e os *middleware* estudados. Assim, foi possível encontrar o RabbitMQ como a plataforma que apresenta melhor desempenho (latência e taxa de transferência de mensagens) nos cenários típicos de comunicação entre agentes.

A solução desenvolvida consiste numa arquitetura que usa o RabbitMQ para a troca de mensagens entre agentes. De acordo com o estudo da literatura feito, esta é a primeira solução que usa o RabbitMQ como *middleware* de comunicação numa plataforma multi-agente. Esta solução permite ainda a distribuição de aplicações/agentes externos por diversas máquinas, mantendo a segurança em todo o processo e aumentando a capacidade de simulação da plataforma. O processo de integração desta solução genérica na plataforma do LIACC é descrito no capítulo 5, podendo ser adaptado para outras plataformas.

Os testes de comparação do antes e depois mostraram o sucesso da arquitetura desenvolvida, permitindo à plataforma do LIACC comunicar agora de forma segura e com melhor desempenho.

Com o uso do RabbitMQ de forma insegura, tal como na solução inicial com o AgentService, obtivemos latências entre 900 e 16.000 vezes mais baixas nos vários cenários testados, bem como taxas de mensagens entre 10 e 120 vezes mais elevadas. Mesmo com o uso de segurança foi possível obter melhorias no desempenho, com latências entre 600 e 8.000 vezes mais baixas nos vários cenários testados, bem como taxas de mensagens entre 10 e 100 vezes mais elevadas.

As questões de investigação, apresentadas no capítulo 1, podem agora agora ser respondidas. Em relação a RQ1 (Será possível encontrar um *middleware* que permite a comunicação segura mantendo um elevado desempenho?), os resultados dos testes de desempenho, desenvolvidos na secção 4.3, bem como os testes do antes e depois no capítulo 6, mostraram que o uso de segurança provoca algum impacto no desempenho apresentado. No contexto da plataforma do LIACC, os testes de avaliação da solução mostram que o uso de segurança no RabbitMQ provoca um aumento da latência de 75% e uma diminuição da taxa de mensagens de 18%. Em alguns cenários esse impacto pode ser demasiado elevado, noutros aceitável, pelo que cada cenário terá de ser avaliado.

Em relação a RQ2 (Será que a integração de um *middleware* num MAS consegue assegurar comunicação mais eficiente e segura?), os resultados obtidos indicam que sim. Os testes de desempenho na secção 4.3 mostraram que, em geral, os *middleware* estudados apresentam melhor desempenho que as plataformas MAS estudadas, sem e com o uso de segurança. Para além disso, os resultados do capítulo 6 mostraram que, depois da integração na plataforma do LIACC, o RabbitMQ manteve melhor desempenho que todas as plataformas MAS estudadas.

Em relação a RQ3 (Será possível o desenvolvimento de uma arquitetura genérica que permite a integração deste *middleware* na plataforma do LIACC, mas também numa qualquer plataforma multi-agente?), acreditamos que a arquitetura desenvolvida na secção 4.4, bem como a descrição detalhada do processo que a permitiu integrar na plataforma do LIACC, disponível no capítulo 5, devem permitir a integração do RabbitMQ numa qualquer plataforma multi-agente, assegurando comunicação eficiente e segura. No entanto, e devido à grande diversidade de plataformas existentes, será necessário avaliar cada caso em concreto.

7.1 Limitações e Trabalho Futuro

A solução desenvolvida apresenta algumas limitações que podem ser abordadas e resolvidas em futuros desenvolvimentos da plataforma.

Em termos de tolerância a falhas, temos de reconhecer que esta arquitetura tem dois pontos centrais em que uma falha pode parar toda a plataforma: o único *broker* do RabbitMQ e o Painel de Controlo. No caso do *broker* do RabbitMQ, se este falhar toda a plataforma perde a capacidade de comunicação e assim o poder de simulação. No entanto, este problema pode ser resolvido com o uso de vários *brokers* em *cluster*[1] para elevada consistência ou em federação[2] para maior disponibilidade. O uso de *clustering* em conjunto com um plugin do RabbitMQ, o *Sharding Plugin*[3],

[1]Mais informação disponível em https://www.rabbitmq.com/clustering.html
[2]Mais informação disponível em https://www.rabbitmq.com/federation.html
[3]Mais informação disponível em https://github.com/rabbitmq/rabbitmq-sharding

permite também melhorar a escalabilidade. O uso do RabbitMQ com vários *brokers* seria também uma mais valia para a melhoria do desempenho da comunicação [Johansson, 2019].

A falha do Painel de Controlo durante o arranque de uma simulação pode levar a que o serviço de certificação do sistema fique também inoperacional. Assim, e assumindo o uso de segurança nessa simulação, nenhum agente consegue obter um certificado assinado pela CA e por sua vez comunicar, parando toda a simulação. Este problema pode ser resolvido com o uso de várias instâncias/nós do Painel de Controlo, usando o mecanismo de *Role Exchange* (Troca de papéis) entre um conjunto de nós *Master* e *Slave* [Gankevich et al., 2016].

No contexto da plataforma do LIACC, o uso de ficheiros com a informação de cenário, equipas, distúrbios e missões implica uma validação desses ficheiros por parte do operador do Painel de Controlo, antes de colocar a simulação em execução. O uso de ficheiros maliciosos pode ser um ponto de entrada na plataforma, contornando o sistema de segurança implementado. Um trabalho futuro poderia criar um sistema automático de validação destes ficheiros, tornando a plataforma mais simples de usar e mais segura.

A arquitectura de segurança produzida permite que um agente recetor consiga verificar o identificador e o endereço IP do agente que lhe enviou uma mensagem. No entanto, este sistema, apesar de seguro, não contempla o processo de decisão do agente sobre se deve confiar ou não no remetente de uma dada mensagem. Teriam de ser encontrados mecanismos que o permitissem fazer de forma segura. Um mecanismo referenciado na literatura é o uso de *distributed trust* (confiança distribuída), em que um agente A vai ajustando o nível de confiança num outro agente B baseado na veracidade da informação que foi recebendo de B [Dorri et al., 2018]. No caso da plataforma do LIACC, poderíamos procurar consenso entre os membros da equipa para decidir se confiam ou não na informação que um outro agente, externo à equipa, envia.

O SimConnect usado para comunicação entre os agentes da plataforma e o FSX utiliza comunicação insegura. Isto permite alterações nas mensagens, como leituras de sensores, por parte de agentes maliciosos, influenciando o comportamento do agente que as recebe. A Microsoft tem programado o lançamento de um novo simulador[4], Microsoft Flight Simulator, mantendo o suporte ao SimConnect que poderá agora receber comunicação segura. Caso contrário, um trabalho futuro poderia introduzir um *wrapper* no SimConnect permitindo comunicação segura com o FSX.

A segurança da solução desenvolvida poderia também ser melhorada com o uso de técnicas de *machine learning* para analisar e detetar padrões nas mensagens, bem como suportar e detetar alguns vetores de ataque. Na literatura encontramos trabalhos semelhantes como [Vashist et al., 2012] e [Arshad et al., 2018].

As operações de gestão dos Nós de Simulação permitem, neste momento, apenas o lançamento remoto de um Nó de Simulação e o envio de agentes para correr num determinado Nó. Um trabalho futuro poderia introduzir suporte para a gestão em tempo real dos diversos Nós de Simulação, permitindo, por exemplo: ver os recursos que cada agente está a gastar em tempo real; parar um agente em especifico; e migrar agentes entre Nós de Simulação. A migração de um agente

[4]Mais informação disponível em `https://www.flightsimulator.com`

implicaria a revogação do certificado atual do agente, e a criação de um novo certificado com o endereço IP da máquina (Nó de Simulação) onde o agente vai correr.

A arquitetura desenvolvida, para a integração do RabbitMQ numa plataforma multi-agente, implica a configuração e arranque de ambos a plataforma multi-agente e o *broker* do RabbitMQ. Este processo, quando comparado com aquele que o utilizador tem numa típica plataforma multi-agente, é mais trabalhoso para o utilizador, dificultando o seu uso. Num trabalho futuro todos os componentes poderiam ser juntos numa única aplicação, numa solução similar à que o JIAC usa para integrar o ActiveMQ.

Na arquitetura atual da plataforma do LIACC, o *Disturbances Manager* é um agente responsável pela gestão, mas também por todo o processamento de distúrbios externos ao FSX, o que pode causar problemas de desempenho com um número elevado de distúrbios. Assim, seria importante a distribuição deste processamento por vários *slave workers* (um *worker* por distúrbio seria uma arquitetura típica), que podem depois ser distribuídos por várias máquinas. Para além disso, o envio das leituras dos sensores aos agentes veículo pode beneficiar desta distribuição. Como demonstrado na secção 6.4, um aumento do paralelismo permite uma maior taxa de envio de mensagens por parte do RabbitMQ. Nessa secção vimos também que o envio das mensagens com as leituras dos sensores poderia beneficiar do uso de várias instâncias de produção do RabbitMQ, pelo que essa melhoria poderia ser também integrada numa nova arquitetura do DM.

Na sua arquitetura atual, a plataforma do LIACC utiliza os seus agentes todos desenvolvidos na mesma linguagem, plataforma MAS e utilizando o mesmo *middleware* de comunicação. Para a evolução da plataforma e de forma a melhorar a sua interoperabilidade, permitindo a possibilidade do desenvolvimento de novos componentes/agentes em tecnologias diferentes, seria importante a introdução de mecanismos que o permitiam. Nesse sentido, e seguindo o que foi sendo feito na literatura para plataformas como o JIAC e o JADE [Soklabi et al., 2013], poderiam ser desenvolvidos *wrappers* que permitam a integração de agentes desenvolvidos noutras plataforma MAS (com diferentes *middleware* e linguagens) na plataforma do LIACC.

Bibliografia

[Abbas and Egerstedt, 2011] Abbas, W. and Egerstedt, M. (2011). Distribution of agents in heterogeneous multiagent systems. In *Proceedings of 50th IEEE Conference on Decision and Control and European Control Conference (CDC-ECC '11), December 12-15, Orlando, FL, USA, 2011*, pages 976–981. DOI: 10.1109/CDC.2011.6160750.

[Ahuja and Mupparaju, 2014] Ahuja, S. and Mupparaju, N. (2014). Performance Evaluation and Comparison of Distributed Messaging Using Message Oriented Middleware. *Computer and Information Science*, 7(4). DOI: 10.5539/cis.v7n4p9.

[Aiyagari et al., 2008] Aiyagari, S., Arrott, M., Atwell, M., Brome, J., Conway, A., Godfrey, R., Greig, R., Hintjens, P., O'Hara, J., Radestock, M., Richardson, A., Ritchie, M., Sadjadi, S., Schloming, R., Shaw, S., Sustrik, M., Trieloff, C., Trieloff, C., and Vinoski, S. (2008). AMQP Advanced Message Queuing Protocol. Disponível em https://www.rabbitmq.com/resources/specs/amqp0-9-1.pdf (acedido em Junho 2020).

[Almeida, 2017] Almeida, J. F. d. S. (2017). Simulation and Management of Environmental Disturbances in Flight Simulator X. Master's thesis, Universidade do Porto.

[Ananda et al., 1995] Ananda, A. L., Teh, H. C., Lee, C. L., and Koh, E. K. (1995). A client-server based application using ASTRA - An asynchronous remote procedure call (RPC) mechanism. *Journal of Microcomputer Applications*, 18(2):95–113. DOI: 10.1016/S0745-7138(05)80021-1.

[Anumba et al., 2005] Anumba, C., Ugwu, O., and Ren, Z. (2005). *Agents and Multi-Agent Systems in Construction*, chapter 3 - Multi-agent systems in contruction: an overview, pages 31–86. Spon Press.

[Apache Software Foundation, 2019a] Apache Software Foundation (2019a). 7. Security. Disponível em https://kafka.apache.org/documentation/#security (acedido em Novembro 2019).

[Apache Software Foundation, 2019b] Apache Software Foundation (2019b). Addressing Model. Disponível em https://activemq.apache.org/components/artemis/documentation/latest/address-model.html (acedido em Novembro 2019).

[Apache Software Foundation, 2019c] Apache Software Foundation (2019c). Consumers. Disponível em https://kafka.apache.org/documentation/#intro_consumers (acedido em Janeiro 2020).

[Apache Software Foundation, 2019d] Apache Software Foundation (2019d). Core. Disponível em https://activemq.apache.org/components/artemis/documentation/latest/core.html (acedido em Novembro 2019).

[Apache Software Foundation, 2019e] Apache Software Foundation (2019e). How does ActiveMQ compare to Artemis. Disponível em https://activemq.apache.org/how-does-activemq-compare-to-artemis (acedido em Novembro 2019).

[Apache Software Foundation, 2019f] Apache Software Foundation (2019f). Introduction. Disponível em https://kafka.apache.org/intro (acedido em Novembro 2019).

[Apache Software Foundation, 2019g] Apache Software Foundation (2019g). Legal Notice. Disponível em https://activemq.apache.org/components/artemis/documentation/1.5.0/notice.html (acedido em Novembro 2019).

[Apache Software Foundation, 2019h] Apache Software Foundation (2019h). Openwire. Disponível em https://activemq.apache.org/components/artemis/documentation/latest/openwire.html (acedido em Novembro 2019).

[Apache Software Foundation, 2019i] Apache Software Foundation (2019i). Quickstart. Disponível em https://kafka.apache.org/quickstart (acedido em Janeiro 2020).

[Apache Software Foundation, 2019j] Apache Software Foundation (2019j). Running the Server. Disponível em http://activemq.apache.org/components/artemis/documentation/1.0.0/running-server.html (acedido em Novembro 2019).

[Apache Software Foundation, 2019k] Apache Software Foundation (2019k). Security. Disponível em https://activemq.apache.org/components/artemis/documentation/latest/security.html (acedido em Novembro 2019).

[Arshad et al., 2018] Arshad, M., Ullah, Z., Ahmad, N., Khalid, M., Criuckshank, H., and Cao, Y. (2018). A survey of local/cooperative-based malicious information detection techniques in VANETs. *EURASIP Journal on Wireless Communications and Networking*, 2018(1):62. DOI: 10.1186/s13638-018-1064-y.

[Barnes et al., 2015] Barnes, R., Thomson, M., Pironti, A., and Langley, A. (2015). Deprecating Secure Sockets Layer Version 3.0. Disponível em https://tools.ietf.org/html/rfc7568 (acedido em Junho 2020).

[Barton, 2020] Barton, J. (2020). Generate and sign certificate in different machines C#. Disponível em https://stackoverflow.com/questions/60930065/generate-and-sign-certificate-in-different-machines-c-sharp (acedido em Junho 2020).

[Bellifemine et al., 2007] Bellifemine, F. L., Caire, G., and Greenwood, D. (2007). *Developing Multi-Agent Systems with JADE (Wiley Series in Agent Technology)*. John Wiley & Sons, Inc., Hoboken, NJ, USA.

[Borselius, 2002] Borselius, N. (2002). Security in multi-agent systems. In *Proceedings of the 2002 International Conference on Security and Management (SAM'02), June, Las Vegas, Nevada, 2002*, pages 31–36.

[Braga et al., 2008] Braga, R., Rossetti, R., Reis, L. P., and Oliveira, E. (2008). Applying multi-agent systems to simulate dynamic control in flexible manufacturing scenarios. In *Proceedings of 19th European Meeting on Cybernetics and Systems Research (EMCSR 2008). March 25-28, Vienna, Austria, 2008*, volume 2, pages 488–493.

[Breakwell, 2008a] Breakwell, J. (2008a). Authenticating MSMQ messages between forests. Disponível em https://blogs.msdn.microsoft.com/johnbreakwell/2008/10/13/authenticating-msmq-messages-between-forests/ (acedido em Dezembro 2019).

[Breakwell, 2008b] Breakwell, J. (2008b). Sending encrypted MSMQ messages. Disponível em https://blogs.msdn.microsoft.com/johnbreakwell/2008/09/12/sending-encrypted-msmq-messages/ (acedido em Dezembro 2019).

[Briones et al., 2016] Briones, A. G., Chamoso, P., and Barriuso, A. (2016). Review of the Main Security Problems with Multi-Agent Systems used in E-commerce Applications. *Advances in Distributed Computing and Artificial Intelligence Journal*, 5(3):55–61. DOI: 10.14201/ADCAIJ2016535561.

[Buchanan, 2017] Buchanan, B. (2017). In An IoT and Quantum World, Is There Still a Place for RSA? ... Boiling a Teaspoon. Disponível em https://www.linkedin.com/pulse/iot-quantum-world-still-place-rsa-boiling-teaspoon-william-buchanan/ (acedido em Janeiro 2020).

[Caire, 2009] Caire, G. (2009). Jade Tutorial - Jade Programming For Beginners. Disponível em https://jade.tilab.com/doc/tutorials/JADEProgramming-Tutorial-for-beginners.pdf (acedido em Janeiro 2020).

[Caire, 2011] Caire, G. (2011). R: [jade-develop] compile leap under .net. Disponível em https://jade.tilab.com/pipermail/jade-develop/2011q2/017099.html (acedido em Junho 2020).

[Caire and Pieri, 2011] Caire, G. and Pieri, F. (2011). Leap User Guide. Disponível em https://jade.tilab.com/doc/tutorials/LEAPUserGuide.pdf (acedido em Junho 2020).

[Calvaresi et al., 2017] Calvaresi, D., Marinoni, M., Sturm, A., Schumacher, M., and Buttazzo, G. (2017). The Challenge of Real-Time Multi-Agent Systems for Enabling IoT and CPS. In *Proceedings of the 2017 International Conference on Web Intelligence (WI'17), August 23-26, Leipzig, Germany, 2017*, pages 356—-364, New York, NY, USA. Association for Computing Machinery. DOI: 10.1145/3106426.3106518.

[Carrascosa et al., 2019] Carrascosa, C., Terrasa, A., Palanca, J., and Julián, V. (2019). SimFleet: A New Transport Fleet Simulator Based on MAS. In De La Prieta, F., González-Briones, A., Pawleski, P., Calvaresi, D., Del Val, E., Lopes, F., Julian, V., Osaba, E., and Sánchez-Iborra, R., editors, *Proceedings of International Workshops of Practical Applications of Survivable Agents and Multi-Agent Systems (PAAMS 2019), June 26-28, Ávila, Spain, 2019*, volume 1047 of *Communications in Computer and Information Science*, pages 257–264, Cham. Springer International Publishing. DOI: 10.1007/978-3-030-24299-2_22.

[Celar et al., 2016] Celar, S., Mudnic, E., and Seremet, Z. (2016). State-of-the-art of messaging for distributed computing systems. In B., K., editor, *Proceedings of 27th International Symposium on Intelligent Manufacturing and Automation (DAAAM'16), 26-29 October, Mostar, Bosnia and Herzegovina, 2016*, volume 27, pages 298–307. Danube Adria Association for Automation and Manufacturing, DAAAM. DOI: 10.2507/27th.daaam.proceedings.044.

[Chalupsky et al., 1992] Chalupsky, H., Finin, T., Fritzson, R., Mckay, D., Shapiro, S., and Wiederhold, G. (1992). An Overview of KQML: A Knowledge Query and Manipulation Language. Technical report, Department of Computer Science, University of Maryland.

[Chappell, 2005] Chappell, D. (2005). Introducing Windows Communication Foundation. Disponível em http://www.davidchappell.com/IntroducingWCFv1.2.1.pdf (acedido em Dezembro 2019).

[Chen et al., 2018] Chen, C., Tock, Y., and Girdzijauskas, S. (2018). BeaConvey: Co-Design of Overlay and Routing for Topic-Based Publish/Subscribe on Small-World Networks. In *Proceedings of the 12th ACM International Conference on Distributed and Event-Based Systems (DEBS '18), June 25-29, Hamilton, New Zealand, 2018*, pages 64—-75. DOI: 10.1145/3210284.3210287.

[CloudFlare, 2020] CloudFlare (2020). How Does Public Key Encryption Work? | Public Key Cryptography and SSL. Disponível em https://www.cloudflare.com/learning/ssl/how-does-public-key-encryption-work/ (acedido em Junho 2020).

[Corsaro et al., 2006] Corsaro, A., Querzoni, L., Scipioni, S., Tucci-Piergiovanni, S., and Virgillito, A. (2006). *Quality of Service in Publish/Subscribe Middleware*, volume 8 of *Emerging Communication: Studies in New Technologies and Practices in Communication*, pages 79–97. IOS Press.

[Curve Project, 2019] Curve Project (2019). Read The Docs. Disponível em http://curvezmq.org/page:read-the-docs (acedido em Novembro 2019).

[DAI-Labor, 2017a] DAI-Labor (2017a). JIAC TNG - Overview. Disponível em https://repositories.dai-labor.de//jiactng/5.2.4/ (acedido em Janeiro 2020).

[DAI-Labor, 2017b] DAI-Labor (2017b). MessageBroker. Disponível em https://repositories.dai-labor.de//jiactng/5.2.4/external-interfaces.html#MessageBroker (acedido em Janeiro 2020).

[DAI-Labor, 2019] DAI-Labor (2019). A Short Introduction to JIAC - Version 5.2.4. Disponível em http://jiac.de/Downloads/jiac/JIAC-Intro.pdf (acedido em Janeiro 2020).

[DAI-Labor, 2020] DAI-Labor (2020). JIAC V. Disponível em http://www.jiac.de/agent-frameworks/jiac-v/ (acedido em Junho 2020).

[Damasceno, 2020] Damasceno, R. R. (2020). Co-Simulation Architecture for Environmental Disturbances. Master's thesis, Universidade do Porto.

[Denis and Johnson, 2007] Denis, T. S. and Johnson, S. (2007). *Cryptography for Developers*, chapter 5 - Hash Functions, pages 203–250. Syngress, Burlington. DOI: 110.1016/B978-159749104-4/50008-X.

[Diffie and Hellman, 1976] Diffie, W. and Hellman, M. (1976). New directions in cryptography. *IEEE Transactions on Information Theory*, 22(6):644–654. DOI: 10.1109/TIT.1976.1055638.

[Dobbelaere and Esmaili, 2017] Dobbelaere, P. and Esmaili, K. S. (2017). Kafka versus RabbitMQ: A Comparative Study of Two Industry Reference Publish/Subscribe Implementations: Industry Paper. In *Proceedings of the 11th ACM International Conference on Distributed and Event-Based Systems (DEBS'17), June 19-23, Barcelona, Spain, 2017*, page 227–238. DOI: 10.1145/3093742.3093908.

[Dorri et al., 2018] Dorri, A., Kanhere, S. S., and Jurdak, R. (2018). Multi-Agent Systems: A Survey. *IEEE Access*, 6:28573–28593. DOI: 10.1109/ACCESS.2018.2831228.

[Eclipse Foundation, 2020] Eclipse Foundation (2020). Eclipse Cyclone DDS 0.6.0 (Florestan). Disponível em https://projects.eclipse.org/projects/iot.cyclonedds/releases/0.6.0-florestan (acedido em Junho 2020).

[Eddelbuettel et al., 2016] Eddelbuettel, D., Stokely, M., and Ooms, J. (2016). RProtoBuf: Efficient Cross-Language Data Serialization in R. *Journal Statistical Software*, 71(2):1–24. DOI: 10.18637/jss.v071.i02.

[Elgamal, 1985] Elgamal, T. (1985). A public key cryptosystem and a signature scheme based on discrete logarithms. *IEEE Transactions on Information Theory*, 31(4):469–472. DOI: 10.1109/TIT.1985.1057074.

[Elkady and Sobh, 2012] Elkady, A. and Sobh, T. (2012). Robotics Middleware: A Comprehensive Literature Survey and Attribute-Based Bibliography. *Journal of Robotics*, 2012. DOI: 10.1155/2012/959013.

[Esteves, 2020] Esteves, J. F. V. D. (2020). Distributed Simulation and Exploration of a Game Environment. Master's thesis, Universidade do Porto.

[Estrada and Astudillo, 2015] Estrada, N. and Astudillo, H. (2015). Comparing scalability of message queue system: ZeroMQ vs RabbitMQ. In *Proceedings of 2015 Latin American Computing Conference (CLEI 2015), October 19-23, Arequipa, Peru, 2015*, pages 1–6. DOI: 10.1109/CLEI.2015.7360036.

[Fatunmbi et al., 2017] Fatunmbi, R., Belkacemi, R., Ariyo, F. K., and Radman, G. (2017). Genetic Algorithm based Optimized Load Frequency Control for Storageless Photo Voltaic Generation in a Two Area Multi-Agent System. In *Proceedings of 2017 North American Power Symposium (NAPS 2017), September 17-19, Morgantown, WV, USA, 2017*, pages 1–5. DOI: 10.1109/NAPS.2017.8107302.

[Feng et al., 2016] Feng, Z., Hu, G., and Wen, G. (2016). Distributed consensus tracking for multi-agent systems under two types of attacks. *International Journal of Robust and Nonlinear Control*, 26(5):896–918. DOI: 10.1002/rnc.3342.

[FIPA TC Agent Management, 2002] FIPA TC Agent Management (2002). Agent Message Transport Reference Model. Standard SC00067F, Foundation for Intelligent Physical Agents.

[FIPA TC Agent Management, 2004] FIPA TC Agent Management (2004). Agent Management Reference Model. Standard SC00023K, Foundation for Intelligent Physical Agents.

[Fischer et al., 2016] Fischer, H., Vulliez, P., Gazeau, J.-P., and Zeghloul, S. (2016). An industrial standard based control architecture for multi-robot real time coordination. In *Proceedings of the IEEE 14th International Conference on Industrial Informatics (INDIN'16), July 19-21, Poitiers, France, 2016*, pages 207–212. DOI: 10.1109/INDIN.2016.7819160.

[Freier et al., 2011] Freier, A., Karlton, P., and Kocher, P. (2011). The Secure Sockets Layer (SSL) Protocol Version 3.0. Disponível em https://tools.ietf.org/html/rfc6101#section-1 (acedido em Janeiro 2020).

[FSDeveloper, 2020] FSDeveloper (2020). SDK Installation (FSX). Disponível em https://www.fsdeveloper.com/wiki/index.php?title=SDK_Installation_(FSX) (acedido em Maio 2020).

[Gankevich et al., 2016] Gankevich, I., Tipikin, Y., Korkhov, V., Gaiduchok, V., Degtyarev, A., and Bogdanov, A. (2016). Factory: Master Node High-Availability for Big Data Applications and Beyond. In Gervasi, O., Murgante, B., Misra, S., Rocha, A. M. A., Torre, C. M., Taniar, D., Apduhan, B. O., Stankova, E., and Wang, S., editors, *Proceedings of 16th International Conference on Computational Science and Its Applications (ICCSA 2016), July 4–7, Beijing, China, 2016*, volume 9787 of *Lecture Notes in Computer Science*, pages 379–389, Cham. Springer International Publishing. DOI: 10.1007/978-3-319-42108-7_29.

[Gelvez García et al., 2019] Gelvez García, N. Y., Ballén Duarte, A. D., and Espitia Cuchango, H. E. (2019). Multi-Agent System Used for Recommendation of Historical and Cultural Memories. *Tecciencia*, 14:43–52. DOI: 10.18180/tecciencia.2019.24.6.

[Ghadimi et al., 2019] Ghadimi, P., Wang, C., Lim, M. K., and Heavey, C. (2019). Intelligent sustainable supplier selection using multi-agent technology: Theory and application for Industry 4.0 supply chains. *Computers & Industrial Engineering*, 127:588–600. DOI: 10.1016/j.cie.2018.10.050.

[Giaimo et al., 2015] Giaimo, F., Andrade, H., Berger, C., and Crnkovic, I. (2015). Improving Bandwidth Efficiency with Self-Adaptation for Data Marshalling on the Example of a Self-Driving Miniature Car. In *Proceedings of the 2015 European Conference on Software Architecture Workshops (ECSAW '15), September 7-11, Dubrovnik, Cavtat, Croatia, 2015*, New York, NY, USA. Association for Computing Machinery. DOI: 10.1145/2797433.2797454.

[Gilchrist, 2016] Gilchrist, A. (2016). *Industry 4.0: The Industrial Internet of Things*, chapter 8: Middleware Software Patterns, pages 131–143. Apress Media, California. DOI: 10.1007/978-1-4842-2047-4.

[Gregori et al., 2006] Gregori, M. E., Cámara, J. P., and Bada, G. A. (2006). A Jabber-Based Multi-Agent System Platform. In *Proceedings of the Fifth International Joint Conference on Autonomous Agents and Multiagent Systems (AAMAS '06), May 8-12, Hakodate, Japan, 2006*, page 1282–1284. DOI: 10.1145/1160633.1160866.

[Grigorik, 2010] Grigorik, I. (2010). ZeroMQ: Modern & Fast Networking Stack. Disponível em https://www.igvita.com/2010/09/03/zeromq-modern-fast-networking-stack/ (acedido em Novembro 2019).

[Hans Van't Hag, 2017] Hans Van't Hag (2017). Comparing Vortex OpenSplice to RabbitMQ. Disponível em https://istkb.adlinktech.com/article/comparing-vortex-opensplice-rabbitmq/ (acedido em Junho 2020).

[Haque et al., 2018] Haque, M. E., Zobaed, S., Islam, M. U., and Areef, F. M. (2018). Performance Analysis of Cryptographic Algorithms for Selecting Better Utilization on Resource Constraint Devices. In *Proceedings of 2018 21st International Conference of Computer and Information Technology (ICCIT'18), 21-23 December, Dhaka, Bangladesh, 2018*. DOI: 10.1109/ICCITECHN.2018.8631957.

[Hedin and Moradian, 2015] Hedin, Y. and Moradian, E. (2015). Security in Multi-Agent Systems. In *Proceedings of 19th Annual Conference on Knowledge-Based and Intelligent Information & Engineering Systems (KES 2015), September 7-9, Singapore, 2015*, volume 60, pages 1604–1612. DOI: 10.1016/j.procs.2015.08.270.

[Henmi, 2006] Henmi, A. (2006). *Firewall Policies and VPN Configurations*, chapter Chapter 5 - Defining a VPN, pages 211–265. Syngress, Burlington. DOI: 10.1016/B978-159749088-7/50007-4.

[Heydari and Effatparvar, 2013] Heydari, R. and Effatparvar, M. (2013). Security in Distributed Multi-Agent Systems. *International Research Journal of Applied and Basic Sciences*, 6:889–896.

[Hintjens, 2008] Hintjens, P. (2008). RestMS - RESTful Messaging Service. Disponível em http://amqp.wikidot.com/wiki:spec-7-v1#toc2 (acedido em Junho 2020).

[Hintjens, 2013] Hintjens, P. (2013). *ZeroMQ: Messaging for Many Applications*. O'Reilly and Associate Series. O'Reilly Media, Incorporated.

[Hirsch et al., 2006] Hirsch, B., Konnerth, T., Heler, A., and Albayrak, S. (2006). A Serviceware Framework for Designing Ambient Services. In *Proceedings of the First International Conference on Ambient Intelligence Developments (AmID'06), September 20-22, Sophia Antipolis, France, 2006*, pages 124–136. Springer, Paris. DOI: 10.1007/978-2-287-47610-5_9.

[Jackson, 2013] Jackson, W. (2013). Why salted hash is as good for passwords as for breakfast. Disponível em https://gcn.com/articles/2013/12/02/hashing-vs-encryption.aspx (acedido em Janeiro 2020).

[JADE Board, 2005] JADE Board (2005). JADE Security Guide. Disponível em https://jade.tilab.com/doc/tutorials/JADE_Security.pdf (acedido em Junho 2020).

[Johansson, 2019] Johansson, L. (2019). Part 2: RabbitMQ Best Practice for High Performance (High Throughput). Disponível em https://www.cloudamqp.com/blog/2018-01-08-part2-rabbitmq-best-practice-for-high-performance.html (acedido em Junho 2020).

[Khegai et al., 2015] Khegai, M., Zubok, D., and Maiatin, A. (2015). Ontology-Based Approach to Scheduling of Jobs Processed by Applications Running in Virtual Environments. In Klinov, P. and Mouromtsev, D., editors, *Proceedings of 6th International Conference on Knowledge Engineering and Semantic Web (KESW 2015), September 30 - October 2, Moscow, Russia, 2015*, volume 518 of *Communications in Computer and Information Science*, pages 273–282, Cham. Springer International Publishing. DOI: 10.1007/978-3-319-24543-0_21.

[Kibble, 2006] Kibble, R. (2006). Speech acts, commitment and multi-agent communication. *Computational and Mathematical Organization Theory*, 12(2–3):127–145. DOI: 10.1007/s10588-006-9540-z.

[Koblitz, 1987] Koblitz, N. (1987). Elliptic curve cryptosystems. *Mathematics of Computation 48 (1987)*, 48(177):203–209. DOI: 10.1090/S0025-5718-1987-0866109-5.

[Konnerth et al., 2012] Konnerth, T., Chinnow, J., Kaiser, S., Grunewald, D., Bsufka, K., and Albayrak, S. (2012). Integration of Simulations and MAS for Smart Grid Management Systems. In *Proceedings of the 3rd International Workshop on Agent Technologies for Energy Systems (ATES 2012), June 5, Valencia, Spain, 2012*, pages 51–58.

[Korneck, 2017] Korneck, C. (2017). Enabling the hidden OpenSSH server in Windows 10 Fall Creators Update (1709) — and why it's great! Disponível em https://poweruser.blog/enabling-the-hidden-openssh-server-in-windows-

10-fall-creators-update-1709-and-why-its-great-51c9d06db8df (acedido em Junho 2020).

[Koschuch et al., 2012] Koschuch, M., Hudler, M., and Krüger, M. (2012). The Price of Security: A Detailed Comparison of the TLS Handshake Performance on Embedded Devices When Using Elliptic Curve Cryptography and RSA. In Obaidat, M. S., Tsihrintzis, G. A., and Filipe, J., editors, *Proceeding of 7th International Joint Conference on e-Business and Telecommunications (ICETE 2010), July 26-28, Athens, Greece, 2010*, pages 71–83, Berlin, Heidelberg. Springer Berlin Heidelberg. DOI: 10.1007/978-3-642-25206-8.

[Kravari and Bassiliades, 2015] Kravari, K. and Bassiliades, N. (2015). A Survey of Agent Platforms. *Journal of Artificial Societies and Social Simulation*, 18(1):18. DOI: 10.18564/jasss.2661.

[Kreps, 2014] Kreps, J. (2014). Benchmarking Apache Kafka: 2 Million Writes Per Second (On Three Cheap Machines). Disponível em https://engineering.linkedin.com/kafka/benchmarking-apache-kafka-2-million-writes-second-three-cheap-machines (acedido em Novembro 2019).

[Küster et al., 2013] Küster, T., Lützenberger, M., and Freund, D. (2013). Distributed Evolutionary Optimisation for Electricity Price Responsive Manufacturing using Multi-Agent System Technology. *International Journal On Advances in Intelligent Systems*, 6(1&2):27–40.

[Lauener and Sliwinski, 2017] Lauener, J. and Sliwinski, W. (2017). How to design & implement a modern communication middleware based on ZeroMQ. In *Proceedings of International Conference on Accelerator and Large Experimental Physics Control Systems (ICALEPCS'17), October 8-13, Barcelona, Spain, 2017*. DOI: 10.18429/JACoW-ICALEPCS2017-MOBPL05.

[Leitão et al., 2013] Leitão, P., Inden, U., and Ruckemann, C.-P. (2013). Parallelising Multi-agent Systems for High Performance Computing. In *Proceedings of Third International Conference on Advanced Communications and Computation (INFOCOMP'13), November 17-22, Lisbon, Portugal, 2013*, pages 1–6.

[Leon et al., 2015] Leon, F., Paprzycki, M., and Ganzha, M. (2015). A Review of Agent Platforms. Technical report, CT COST Action IC1404, Multi-Paradigm Modelling for Cyber-Physical Systems (MPM4CPS).

[Li et al., 2018] Li, Y., Frasure, I., Ikusan, A. A., Zhang, J., and Dai, R. (2018). Vulnerability Assessment for Unmanned Systems Autonomy Services Architecture. In Au, M. H., Yiu, S. M., Li, J., Luo, X., Wang, C., Castiglione, A., and Kluczniak, K., editors, *Proceedings of 12th International Conference on Network and System Security (NSS 2018), August 27-29, Hong Kong, China, 2018*, volume 11058 of *Lecture Notes in Computer Science*, pages 266–276, Cham. Springer International Publishing. DOI: 10.1007/978-3-030-02744-5_20.

[Liu et al., 2020] Liu, X., Zhang, T., Hu, N., Zhang, P., and Zhang, Y. (2020). The method of Internet of Things access and network communication based on MQTT. *Computer Communications*, 153:169–176. DOI: 10.1016/j.comcom.2020.01.044.

[Lopes Cardoso et al., 2016] Lopes Cardoso, H., Urbano, J., Rocha, A. P., Castro, A. J. M., and Oliveira, E. (2016). *ANTE: A Framework Integrating Negotiation, Norms and Trust*, volume 30 of *Law, Governace and Technology*, pages 27–45. Springer International Publishing, Cham. DOI: 10.1007/978-3-319-33570-4_3.

[Luzuriaga et al., 2015a] Luzuriaga, J., Boronat, P., Perez, M., Calafate, C., Cano, J.-C., and Manzoni, P. (2015a). A comparative evaluation of AMQP and MQTT protocols over unstable and mobile networks. In *Proceedings of 12th Annual IEEE Consumer Communications and Networking Conference (CCNC 2015), January 9-12, Las Vegas, USA, 2015*, pages 931–936. DOI: 10.1109/CCNC.2015.7158101.

[Luzuriaga et al., 2015b] Luzuriaga, J., Boronat, P., Perez, M., Calafate, C., Cano, J.-C., and Manzoni, P. (2015b). A comparative evaluation of AMQP and MQTT protocols over unstable and mobile networks. In *Proceedings of 2015 12th Annual IEEE Consumer Communications and Networking Conference (CCNC 2015), January 9-12, Las Vegas, NV, USA, 2015*, pages 931–936. DOI: 10.1109/CCNC.2015.7158101.

[Lützenberger et al., 2015] Lützenberger, M., Konnerth, T., and Küster, T. (2015). *Industrial Agents*, chapter 21 - Programming of Multiagent Applications with JIAC, pages 381–398. Morgan Kaufmann, Boston. DOI: 10.1016/B978-0-12-800341-1.00021-8.

[Mahmoud, 2004] Mahmoud, Q. (2004). Getting Started with Java Message Service (JMS). Technical report, Oracle.

[Marcos, 2016] Marcos, P. B. (2016). Plataforma de Monitorização de Recursos num Sistema Message Oriented Middleware. Master's thesis, Universidade do Porto.

[Mesfin et al., 2018] Mesfin, G., Ghinea, G., Grønli, T.-M., and Younas, M. (2018). Web Service Composition on Smartphones: The Challenges and a Survey of Solutions. In Younas, M., Awan, I., Ghinea, G., and Catalan Cid, M., editors, *Proceeding of 15th International Conference on Mobile Web and Intelligent Information Systems (MobiWIS 2018), August 6–8, Barcelona, Spain, 2018*, pages 126–141, Cham. Springer International Publishing. DOI: 10.1007/978-3-319-97163-6.

[Michel et al., 2009] Michel, F., Ferber, J., and Drogoul, A. (2009). Multi-Agent Systems and Simulation: a Survey From the Agents Community's Perspective. In Danny Weyns, A. U., editor, *Multi-Agent Systems: Simulation and Applications*, Computational Analysis, Synthesis, and Design of Dynamic Systems, pages 3–53. CRC Press - Taylor & Francis. DOI: 10.1201/9781420070248.pt1.

[Microsoft, 2008] Microsoft (2008). SimConnect SDK Reference. Disponível em https://docs.microsoft.com/en-us/previous-versions/microsoft-esp/cc526983%28v%3dmsdn.10%29 (acedido em Junho 2020).

[Miller, 1985] Miller, V. S. (1985). Use of Elliptic Curves in Cryptography. In *Proceedings of Advances in Cryptology (CRYPTO'85), August 18-22, Santa Barbara, California, USA, 1985*, page 417–426.

[Mohamed et al., 2008] Mohamed, N., Al-Jaroodi, J., and Jawhar, I. (2008). Middleware for Robotics: A Survey. In *Proceedings of 2008 IEEE Conference on Robotics, Automation and Mechatronics, 21-24 September, Chengdu, China, 2008*, pages 736–742. DOI: 10.1109/RAMECH.2008.4681485.

[Mollin, 2002] Mollin, R. A. (2002). *RSA and Public-Key Cryptography*. Discrete Mathematics and Its Applications. CRC Press, Inc., USA.

[Moreno et al., 2003] Moreno, A., Sánchez, D., and Isern, D. (2003). Security Measures in a Medical Multi-Agent System. In *Proceedings of the 6th International Conference of the Catalan Association for Artificial Intelligence (CCIA 2003), October, 2003, Palma de Mallorca, Spain*, volume 100, pages 244–255.

[Moriarty and Farrell, 2020] Moriarty, K. and Farrell, S. (2020). Deprecating TLSv1.0 and TLSv1.1. Disponível em https://tools.ietf.org/html/draft-ietf-tls-oldversions-deprecate-06 (acedido em Junho 2020).

[Mrozek et al., 2014a] Mrozek, D., Małysiak-Mrozek, B., Mikołajczyk, J., and Kozielski, S. (2014a). *Man-Machine Interactions 3*, volume 242 of *Advances in Intelligent Systems and Computing*, chapter Database Under Pressure – Testing Performance of Database Systems Using Universal Multi-Agent Platform, pages 631–641. Springer International Publishing, Cham. DOI: 10.1007/978-3-319-02309-0_68.

[Mrozek et al., 2014b] Mrozek, D., Małysiak-Mrozek, B., and Waligóra, I. (2014b). UMAP - A Universal Multi-Agent Platform for .NET Developers. In Kozielski, S., Mrozek, D., Kasprowski, P., Małysiak-Mrozek, B., and Kostrzewa, D., editors, *Proceedings of 10th International Conference on Beyond Databases, Architectures, and Structures (BDAS 2014), May 27-30, Ustron, Poland, 2014*, volume 424 of *Communications in Computer and Information Science*, pages 300–311, Cham. Springer International Publishing. DOI: 10.1007/978-3-319-06932-6.

[Nascimento et al., 2017] Nascimento, N. M., Viana, C. J., von Staa, A., and Lucena, C. (2017). A Publish-Subscribe based Architecture for Testing Multiagent Systems. In *Proceeding of 29th International Conference on Software Engineering and Knowledge Engineering (SEKE'17), July 5-7, Pittsburgh, PA, USA, 2017*, pages 521–526. DOI: 10.18293/SEKE2017-050.

[Ning, 2011] Ning, P. (2011). Topic 4. Cryptographic Hash Functions. Department of Computer Science, North Carolina State University. Disponível em http://discovery.csc.ncsu.edu/Courses/csc574-F09/slides/T04_HashFunctions.3pp.pdf (acedido em Dezembro 2019).

[Obitko, 2007] Obitko, M. (2007). *Translations between Ontologies in Multi-Agent Systems*. PhD thesis, Czech Technical University in Prague, Prague, Czech Republic.

[Object Management Group, 2015] Object Management Group (2015). Data Distribution Service (DDS) – Version 1.4. Standard formal/2015-04-10, Object Management Group.

[OpenDDS Project, 2020] OpenDDS Project (2020). OpenDDS. Disponível em https://github.com/objectcomputing/OpenDDS (acedido em Junho 2020).

[Oprea, 2004] Oprea, M. (2004). Applications of Multi-Agent Systems. In Reis, R., editor, *Proceedings of International Federation for Information Processing 18th World Computer Congress (WCC2004), August 22-27, Toulouse, France, 2004*, volume 157 of *IFIP Advances in Information and Communication Technology*, pages 239–270, Boston, MA. Springer US. DOI: 10.1007/1-4020-8159-6_9.

[Palanca, 2019a] Palanca, J. (2019a). Presence Notification. Disponível em https://spade-mas.readthedocs.io/en/latest/presence.html (acedido em Novembro 2019).

[Palanca, 2019b] Palanca, J. (2019b). SPADE. Disponível em https://spade-mas.readthedocs.io/en/latest/readme.html (acedido em Junho 2020).

[Palanca, 2019c] Palanca, J. (2019c). Welcome to SPADE's documentation! Disponível em https://spade-mas.readthedocs.io/en/latest/index.html (acedido em Janeiro 2020).

[Paletta, 2012] Paletta, M. (2012). Self-Organizing Multi-Agent Systems by means of Scout Movement. *Recent Patents on Computer Science*, 5(3):197–210. DOI: 10.2174/2212375911205030197.

[Passadore et al., 2008] Passadore, A., Grosso, A., Coccoli, M., and Boccalatte, A. (2008). Agent-Service in a hand. In *Proceedings of the 9th Workshop "From Objects to Agents"(WOA'08), November 17-18, Palermo, Italy, 2008 - Evolution of Agent Development: Methodologies, Tools, Platforms and Languages*, pages 19–27.

[Peres et al., 2016] Peres, R. S., Rocha, A. D., Coelho, A., and Barata Oliveira, J. (2016). A Highly Flexible, Distributed Data Analysis Framework for Industry 4.0 Manufacturing Systems. In Borangiu, T., Trentesaux, D., Thomas, A., Leitão, P., and Oliveira, J. B., editors, *Proceedings of 6th International Workshop on Service Orientation in Holonic and Multi-Agent Manufacturing (SOHOMA 2016), October 6-7, Lisbon, Portugal, 2016*, volume 694 of *Studies in Computational Intelligence*, pages 373–381, Cham. Springer International Publishing. DOI: 10.1007/978-3-319-51100-9_33.

[Pieter Hintjens, 2009] Pieter Hintjens (2009). RESTful Messaging Service. Disponível em https://github.com/imatix/restms/tree/master (acedido em Junho 2020).

[Pires et al., 2018] Pires, F., Barbosa, J., and Leitão, P. (2018). Quo Vadis Industry 4.0: An Overview Based on Scientific Publications Analytics. In *Proceedings of 2018 IEEE 27th International Symposium on Industrial Electronics (ISIE 2018), June 13-15, Cairns, QLD, Australia, 2018*, pages 663–668. DOI: 10.1109/ISIE.2018.8433868.

[Pivotal RabbitMQ, 2019a] Pivotal RabbitMQ (2019a). Mozilla Public License. Disponível em https://www.rabbitmq.com/mpl.html (acedido em Janeiro 2020).

[Pivotal RabbitMQ, 2019b] Pivotal RabbitMQ (2019b). What can RabbitMQ do for you? Disponível em https://www.rabbitmq.com/features.html (acedido em Novembro 2019).

[Pivotal RabbitMQ, 2020a] Pivotal RabbitMQ (2020a). Authorisation: How Permissions Work. Disponível em https://www.rabbitmq.com/access-control.html#authorisation (acedido em Maio 2020).

[Pivotal RabbitMQ, 2020b] Pivotal RabbitMQ (2020b). Management Plugin. Disponível em https://www.rabbitmq.com/management.html (acedido em Maio 2020).

[Pivotal RabbitMQ, 2020c] Pivotal RabbitMQ (2020c). Manually Generating a CA, Certificates and Private Keys. Disponível em https://www.rabbitmq.com/ssl.html#manual-certificate-generation (acedido em Maio 2020).

[Pivotal RabbitMQ, 2020d] Pivotal RabbitMQ (2020d). Validated User-ID. Disponível em https://www.rabbitmq.com/validated-user-id.html (acedido em Maio 2020).

[Pivotal RabbitMQ, 2020e] Pivotal RabbitMQ (2020e). Which protocols does RabbitMQ support? Disponível em https://www.rabbitmq.com/protocols.html (acedido em Junho 2020).

[Preisler et al., 2016] Preisler, T., Dethlefs, T., and Renz, W. (2016). DeCoF: A Decentralized Coordination Framework for Various Multi-Agent Systems. In Klusch, M., Unland, R., Shehory, O., Pokahr, A., and Ahrndt, S., editors, *Proceedings of 4th German Conference on Multiagent System Technologies (MATES'16), September 27–30, Klagenfurt, Austria, 2016*, volume 9872 of *Lecture Notes in Computer Science*, pages 73–88, Cham. Springer International Publishing. DOI: 10.1007/978-3-319-45889-2_6.

[Pédron et al., 2020] Pédron, J.-S., Software, P., and Klishin, M. (2020). x509 (TLS/SSL) certificate Authentication Mechanism for RabbitMQ. Disponível em https://github.com/rabbitmq/rabbitmq-auth-mechanism-ssl/blob/master/README.md (acedido em Maio 2020).

[Querzoni, 2012] Querzoni, L. (2012). Publish/Subscribe Systems. Disponível em https://www.dis.uniroma1.it/~querzoni/corsi_assets/1112/SistemiDistribuiti/11-Pubsub.pdf (acedido em Novembro 2019).

[Rao, 2019] Rao, J. (2019). Clients. Disponível em https://cwiki.apache.org/confluence/display/KAFKA/Clients (acedido em Novembro 2019).

[Rescorla, 2018] Rescorla, E. (2018). The Transport Layer Security (TLS) Protocol Version 1.3. Disponível em https://tools.ietf.org/html/rfc8446 (acedido em Junho 2020).

[RestMS, 2008] RestMS (2008). RestMS. Disponível em http://www.restms.org/ (acedido em Junho 2020).

[Rivest et al., 1978] Rivest, R. L., Shamir, A., and Adleman, L. (1978). A Method for Obtaining Digital Signatures and Public-Key Cryptosystems. *Communications of the ACM*, 21(2):120–126. DOI: 10.1145/359340.359342.

[Saghian and Ravanmehr, 2014] Saghian, M. and Ravanmehr, R. (2014). A Survey on Middleware Approaches for Distributed Real-Time Systems. *Journal of Mobile, Embedded and Distributed Systems*, 6(4):147–158.

[Saint-Andre, 2020] Saint-Andre, P. (2020). 5. Exchanging Presence Information. Disponível em https://xmpp.org/rfcs/rfc3921.html#presence (acedido em Junho 2020).

[Santos, 2006] Santos, C. A. (2006). Segurança em Web Services. Disponível em http://wiki.di.uminho.pt/twiki/pub/Education/Criptografia/CriptografiaMestrados0607/SegurancaWS.pdf (acedido em Dezembro 2019).

[Shashaj et al., 2019] Shashaj, A., Mastrorilli, F., Morrelli, M., Pansini, G., Iannucci, E., and Polito, M. (2019). A Distributed Multi-Agent System (MAS) Application For continuous and Integrated Big Data Processing. In Chatzigiannakis, I., De Ruyter, B., and Mavrommati, I., editors, *Proceedings of 15th European Conference on Ambient Intelligence (AmI 2019), November 13–15, Rome, Italy, 2019*, volume 11912 of *Lecture Notes in Computer Science*, pages 350–356, Cham. Springer International Publishing. DOI: 10.1007/978-3-030-34255-5_27.

[Shen et al., 2010] Shen, X., Yu, H., and Buford, J. (2010). *Handbook of Peer-to-Peer Networking*. Springer Publishing Company, Incorporated, New York, 1st edition. DOI: 10.5555/1734068.

[Silva, 2011] Silva, D. C. (2011). *Cooperative Multi-Robot Missions: Development of a Platform and a Specification Language*. PhD thesis, University of Porto, Porto, Portugal.

[Soklabi et al., 2013] Soklabi, A., Bahaj, M., and Cherti, I. (2013). JIAC Systems and JADE Agents Communication. *International Journal of Engineering and Technology*, 5(2):1976–1984.

[Sotolár, 2013] Sotolár, O. (2013). .Net Remoting Security. Bachelor thesis, Masaryk University, Faculty of Informatics, Brno, Czech Republic.

[Stan Schneider, 2013] Stan Schneider (2013). What's The Difference Between DDS And AMQP? Disponível em https://www.electronicdesign.com/technologies/embedded-revolution/article/21796186/whats-the-difference-between-dds-and-amqp (acedido em Junho 2020).

[STOMP, 2012a] STOMP (2012a). Protocol Overview. Disponível em https://stomp.github.io/stomp-specification-1.2.html#Overview (acedido em Junho 2020).

[STOMP, 2012b] STOMP (2012b). STOMP 1.2 Implementations. Disponível em https://stomp.github.io/implementations.html (acedido em Junho 2020).

[Subburaj and Urban, 2018] Subburaj, V. H. and Urban, J. E. (2018). Specifying Security Requirements in Multi-agent Systems Using the Descartes-Agent Specification Language and AUML. In Ziemba, E., editor, *Proceedings of 15th Conference on Advanced Information Technologies for Management (AITM 2018) and 13th Conference on Information Systems Management (ISM 2018), September 9-12, Poznan, Poland, 2018*, volume 346 of *Lecture Notes in Business Information Processing*, pages 93–111, Cham. Springer International Publishing. DOI: 10.1007/978-3-030-15154-6_6.

[Sujil et al., 2018] Sujil, A., Verma, J., and Kumar, R. (2018). Multi agent system: concepts, platforms and applications in power systems. *Artificial Intelligence Review*, 49(2):153–182. DOI: 10.1007/s10462-016-9520-8.

[Sullivan, 2013] Sullivan, N. (2013). A (Relatively Easy To Understand) Primer on Elliptic Curve Cryptography. Disponível em https://blog.cloudflare.com/a-relatively-easy-to-understand-primer-on-elliptic-curve-cryptography/ (acedido em Janeiro 2020).

[Szydlo et al., 2017] Szydlo, T., Nawrocki, P., Brzoza-Woch, R., and Zielinski, K. (2017). Power Aware MOM for Telemetry-Oriented Applications—Levee Monitoring Use Case. In Grzenda, M., Awad, A. I., Furtak, J., and Legierski, J., editors, *Advances in Network Systems. iNetSApp 2015*, volume 461 of *Advances in Intelligent Systems and Computing*, pages 279–295, Cham. Springer International Publishing. DOI: 10.1007/978-3-319-44354-6.

[Tangod and Kulkarni, 2018] Tangod, K. and Kulkarni, G. (2018). Secure Communication through MultiAgent System-Based Diabetes Diagnosing and Classification. *Journal of Intelligent Systems*, 29(1):703–718. DOI: 10.1515/jisys-2017-0353.

[Telecom IT, 2017] Telecom IT (2017). JADE. Disponível em https://jade.tilab.com/ (acedido em Novembro 2019).

[The AgentService Team, 2008] The AgentService Team (2008). AgentService External Runtime. Disponível em https://web.archive.org/web/20130829063742/http://www.agentservice.it/download/AgentServiceExternalRuntime.pdf (acedido em Dezembro 2019).

[Trucco et al., 2010] Trucco, T., Caire, G., Bellifemine, F., and Rimassa, G. (2010). Jade Tutorial - Jade Programmer's Guide. Disponível em https://jade.tilab.com/doc/programmersguide.pdf (acedido em Janeiro 2020).

[Uk et al., 2018] Uk, B., Konam, D., Passot, C., Erdelj, M., and Natalizio, E. (2018). Implementing a System Architecture for Data and Multimedia Transmission in a Multi-UAV System. In Chowdhury, K. R., Di Felice, M., Matta, I., and Sheng, B., editors, *Proceedings of 16th International Conference on Wired/Wireless Internet Communication (WWIC 2018), June 18–20, Boston, MA, USA, 2018*, volume 10866 of *Lecture Notes in Computer Science*, pages 246–257, Cham. Springer International Publishing. DOI: 10.1007/978-3-030-02931-9_20.

[Vashist et al., 2012] Vashist, A., Chadha, R., Kaplan, M., and Moeltner, K. (2012). Detecting communication anomalies in tactical networks via graph learning. In *Proceedings of 31st IEEE Military Communications Conference (MILCOM 2012), October 29 - November 1, Orlando, FL, USA, 2012*, pages 1–6. IEEE. DOI: 10.1109/MILCOM.2012.6415763.

[Vecchiola et al., 2008] Vecchiola, C., Grosso, A., and Boccalatte, A. (2008). AgentService: A Framework to Develop Distributed Multiagent Systems. *International Journal of Agent-Oriented Software Engineering*, 2(3):290–323. DOI: 10.1504/IJAOSE.2008.019421.

[Vecchiola et al., 2009] Vecchiola, C., Grosso, A., Passadore, A., and Boccalatte, A. (2009). Agentservice: A Framework for Distributed Multiagent System Development. *International Journal of Computers and Applications*, 31(3):204–210. DOI: 10.2316/Journal.202.2009.3.202-2968.

[Vila et al., 2007] Vila, X., Schuster, A., and Riera, A. (2007). Security for a Multi-Agent System based on JADE. *Computers & Security*, 26(5):391–400. DOI: 10.1016/j.cose.2006.12.003.

[Vojkovic, 2014] Vojkovic, Z. (2014). ZeroMQ Sockets. Disponível em https://speakerdeck.com/zdeslav/zeromq-toolkit-for-high-performance-distributed-applications?slide=8 (acedido em Dezembro 2019).

[Vortex OpenSplice, 2019] Vortex OpenSplice (2019). 11. DDS Security. Disponível em http://download.prismtech.com/docs/Vortex/html/ospl/DeploymentGuide/security.html (acedido em Junho 2020).

[Wang et al., 2013] Wang, V., Salim, F., and Moskovits, P. (2013). *The Definitive Guide to HTML5 WebSocket*, chapter 5: Using Messaging over WebSocket with STOMP, pages 85–108. Apress Media, Berkeley, California. DOI: 10.1007/978-1-4302-4741-8.

[Wang and Zhang, 2012] Wang, X. and Zhang, L. (2012). Multi-Agent Systems Simulation Base on HLA Framework. In Lee, G., editor, *Proceedings of International Conference on Automation and Robotics (ICAR 2011), December 1-2, Dubai, United Arab Emirates, 2011*, volume 2 of *Lecture Notes in Electrical Engineering*, pages 339–346, Berlin, Heidelberg. Springer. DOI: 10.1007/978-3-642-25646-2.

[Wenzel et al., 2020] Wenzel, M., Schonning, N., Youssef, V., Latham, L., Wagner, B., Mabee, D., Dykstra, T., and Blome, Mike, K. A. (2020). Serialization (C#). Disponível em https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/concepts/serialization/ (acedido em Maio 2020).

[Wrock, 2016] Wrock, M. (2016). Certificate (password-less) based authentication in WinRM. Disponível em http://www.hurryupandwait.io/blog/certificate-password-less-based-authentication-in-winrm (acedido em Junho 2020).

[Xie and Liu, 2017] Xie, J. and Liu, C.-C. (2017). Multi-agent systems and their applications. *Journal of International Council on Electrical Engineering*, 7(1):188–197. DOI: 10.1080/22348972.2017.1348890.

[XMPP, 2016] XMPP (2016). An Overview of XMPP. Disponível em https://xmpp.org/about/technology-overview.html (acedido em Novembro 2019).

[XMPP, 2020] XMPP (2020). An Overview of XMPP oftware. Disponível em https://xmpp.org/software/ (acedido em Junho 2020).

[Zak et al., 2012] Zak, J., Horacek, J., Zboril, F., Koci, R., and Kral, J. (2012). JADE agents used for Wireless Sensors control: System based on services. In *Proceedings of 2012 12th International Conference on Intelligent Systems Design and Applications (ISDA'12), November 27-29, Kochi, India, 2012*, pages 252–257. DOI: 10.1109/ISDA.2012.6416546.

[Zeilenga, 2006] Zeilenga, K. (2006). Lightweight Directory Access Protocol (LDAP): String Representation of Distinguished Names. Disponível em https://tools.ietf.org/html/rfc4514 (acedido em Maio 2020).

[ZeroMQ Team, 2019a] ZeroMQ Team (2019a). The Dynamic Discovery Problem. Disponível em http://zguide.zeromq.org/page:all#header-39 (acedido em Janeiro 2020).

[ZeroMQ Team, 2019b] ZeroMQ Team (2019b). ØMQ - The Guide. Disponível em http://zguide.zeromq.org/page:all#toc3 (acedido em Novembro 2019).

[Zhao et al., 2016] Zhao, D., Qiao, K., Zhou, Z., Li, T., Zhou, X., and Raicu, I. (2016). Exploiting Multi-Cores for Efficient Interchange of Large Messages in Distributed Systems. *Concurrency and Computation: Practice and Experience*, 28(13):3568–3585. DOI: 10.1002/cpe.3742.

[Zyre Project, 2020] Zyre Project (2020). Zyre - Local Area Clustering for Peer-to-Peer Applications. Disponível em https://github.com/zeromq/zyre (acedido em Junho 2020).

Anexo A

Algoritmos de Encriptação de Chave Pública

A encriptação de chave pública é a base de vários algoritmos de encriptação como DSA, ElGamal, RSA, criptografia de curva elíptica, entre outros [Haque et al., 2018]. São aqui apresentados em detalhe a criptografia de curva elíptica e o RSA.

A.1 Eliptic Curve

Este algoritmo baseia-se na estrutura algébrica de curvas elípticas sobre corpos finitos, que satisfazem a equação [Koblitz, 1987]:

$$Y^2 = x^3 + ax + b \tag{A.1}$$

Neste algoritmo, os parâmetros a e b que permitem formar a curva são públicos e partilhados, bem como um ponto da curva, chamado ponto gerador, G. Este gerador G, tem uma grande ordem prima n, tal que $n * G = O$, em que O é o elemento identidade. A chave privada é simplesmente um número N, menor que n. A chave publica pode depois ser gerada como sendo [Buchanan, 2017]:

$$P = N * G \tag{A.2}$$

Esta operação de soma do ponto G, N vezes, não segue o processo típico de soma de pontos. A resolução de forma gráfica e mais intuitiva é a seguinte: dados 2 pontos, A e B, para obter a sua soma é necessário traçar uma linha reta que passa por ambos e vai intersetar a curva num terceiro ponto, R. Para o caso especial de A=B, como neste caso, o ponto R seria obtido pela interseção entre a tangente ao ponto A e a curva. O ponto resultante da soma de A e B tem coordenadas (x(R), -y(R)) [Miller, 1985]. Mesmo com o conhecimento dos parâmetros a e b da curva, do ponto G e do número de bits da chave, é extremamente difícil descobrir qual o número de somas que deram origem ao ponto P.

A operação de encriptação de uma mensagem implica a codificação da mensagem como um ponto da curva, usando uma qualquer codificação. Para encriptação usando chave publica e assumindo a comunicação de A para B, A precisa de gerar um número aleatório k e assim produzir o seguinte par de pontos:

$$C_1 = k * G \;\; e \;\; C_2 = P_m + k * P_B \tag{A.3}$$

em que P_m é a mensagem codificada como um ponto, sendo essa disfarçada através da soma ponto gerado por k somas do ponto P_B (chave pública de B).

Para a desencriptação, B precisa de utilizar a sua chave privada (n_B), conseguindo descobrir a mensagem codificada como o ponto P_m:

$$P_m = C_2 - n_B * C_1 \tag{A.4}$$

A.2 RSA

Este algoritmo baseia-se na dificuldade prática da fatorização do produto de dois números primos muito grandes. As chaves são representadas por conjuntos de números, a chave privada por (e, n) e a chave pública por (d, n). O valor n é obtido pela multiplicação de dois números primos muito grandes, podendo estes ser gerados como referido no capítulo "How to Find Large Prime Numbers"de [Rivest et al., 1978].

$$n = p * q \tag{A.5}$$

O valor de d pode ser obtido como o valor tal que:

$$gcd(d, (p-1) * (q-1)) = 1 \tag{A.6}$$

Quanto ao valor de e, este pode ser obtido como o valor tal que:

$$e * d \equiv 1 \; mod((p-1) * (q-1)) \tag{A.7}$$

Para a encriptação de uma mensagem, esta terá de ser transformada em valores numéricos entre 0 e n-1. Uma mensagem (M) pode ser encriptada, gerando a mensagem cifrada (C), usando a chave privada (e), da seguinte forma:

$$C = M^e mod(n) \tag{A.8}$$

Para a desencriptação da mensagem cifrada (C), a mesma operação de exponenciação é usada, mas com a chave pública (d) que reverte o efeito de aplicação da chave privada:

$$M = C^d mod(n) \tag{A.9}$$

Anexo B

Algoritmos de Troca de Chaves

A encriptação de chave simétrica implica um conhecimento prévio da chave partilhada ou que a troca seja feita no início da comunicação. A troca de chaves no início da comunicação tem de garantir que, apesar de não haver ainda confidencialidade nas mensagens que são trocadas, apenas as entidades que comunicam podem no final conhecer a chave que será utilizada. Existem alguns algoritmos que permitem fazê-lo, sendo aqui apresentados o algoritmo de Diffie-Hellman e o algoritmo de Elgamal.

B.1 Diffie-Hellman

Neste algoritmo as entidades que pretendem comunicar de forma segura, digamos A e B, precisam de chegar a um acordo sobre dois números, um número primo q e a sua raiz primitiva[1] α. Estes valores podem ser conhecidos inclusive por atacantes. A e B podem agora gerar duas chaves, uma pública e outra secreta partilhada. Inicialmente geram-se as chaves publicas, a partir de um número secreto. A e B escolhem, cada um, um número inferior a q que será a base das suas chaves, X_A e X_B respetivamente. As chaves públicas são então geradas, para A temos $Y_A = \alpha^{X_A} \ mod \ q$ e para B temos $Y_B = \alpha^{X_B} \ mod \ q$. Estas chaves são partilhas sem qualquer tipo de confidencialidade, mas sem que isso seja um problema. Agora ambos vão chegar à chave partilhada, mas de formas diferentes: A calcula K como sendo $Y_B^{X_A} \ mod \ q$ e B calcula-o como sendo $Y_A^{X_B} \ mod \ q$, ou seja, utilizando a chave pública do outro e o seu número secreto [Diffie and Hellman, 1976].

Este algoritmo apresenta, no entanto, um problema conhecido. Ao utilizar chaves públicas geradas na hora, não consegue garantir que estamos realmente a falar com o agente que pretendemos, estando vulnerável a ataques de *man-in-the-middle*. Um agente malicioso C pode gerar duas sessões, uma entre A e C e outra entre C e B, funcionando assim como intermediário para todas as mensagens trocadas. O agente malicioso pode então capturar o conteúdo das mensagens trocadas, alterá-lo ou descartar mensagens [Diffie and Hellman, 1976].

[1] Mais informação disponível em https://en.wikipedia.org/wiki/Primitive_root_modulo_n

B.2 Elgamal

Este algoritmo tem por base o Diffie-Hellman, mas utiliza a infraestrutura de chave pública para garantir a identidade dos intervenientes. Neste algoritmo, um dos intervenientes tem de tomar a iniciativa de iniciar a comunicação, vamos assumir que foi B. B precisa então de gerar um número secreto, k, entre 0 e $q-1$, sendo q um número primo acordado entre A e B tal como no algoritmo Diffie-Hellman. B vai depois gerar a chave partilhada que A e B vão usar para comunicar, $K = Y_A^k \ mod \ q$. De notar a utilização da chave pública de A, Y_A, na geração da chave partilhada. $Y_A = \alpha_A^X$ com X_A a ser a chave privada de A. Agora, de forma a partilhar esta chave com A, sem um canal de comunicação seguro e para que apenas A consiga perceber a chave, B envia $C = \alpha^k \ mod \ q$. A como sendo o único com conhecimento da sua chave privada, consegue elevar C à sua chave privada e obter K, a chave partilhada entre ambos [Elgamal, 1985].

$$C_A^X = (\alpha^k)_A^X = Y_A^k \ mod \ q = K \tag{B.1}$$

A troca de chaves de forma segura permite, assim, a utilização posterior de encriptação simétrica que é bastante mais rápida de computar e nos garante os mesmo níveis de segurança.