UNIVERSIDADE TECNOLÓGICA FEDERAL DO PARANÁ

PEDRO WARMLING BOTELHO

DESENVOLVIMENTO DE UM PROTÓTIPO CIBER-FÍSICO BASEADO EM AGENTES PARA UM *SMART PARKING*

PONTA GROSSA
2021

PEDRO WARMLING BOTELHO

DESENVOLVIMENTO DE UM PROTÓTIPO CIBER-FÍSICO BASEADO EM AGENTES PARA UM *SMART PARKING*

Development of an agent based cyber-physical prototype for a smart parking

Trabalho de conclusão de curso apresentado como requisito para obtenção do título de Bacharel em Ciência da Computação da Universidade Tecnológica Federal do Paraná (UTFPR).
Orientador(a): Prof. Dr. André Pinz Borges
Coorientador(a): Prof. Dr. Gleifer Vaz Alves

PONTA GROSSA

2021

4.0 Internacional

Esta licença permite remixe, adaptação e criação a partir do trabalho, para fins não comerciais, desde que sejam atribuídos créditos ao(s) autor(es) e que licenciem as novas criações sob termos idênticos. Conteúdos elaborados por terceiros, citados e referenciados nesta obra não são cobertos pela licença.

PEDRO WARMLING BOTELHO

DESENVOLVIMENTO DE UM PROTÓTIPO CIBER-FÍSICO BASEADO EM AGENTES PARA UM SMART PARKING

Trabalho de Conclusão de Curso de Graduação apresentado como requisito para obtenção do título de Bacharel em Ciência da Computação da Universidade Tecnológica Federal do Paraná (UTFPR).

Data de aprovação: 26 de novembro de 2021

André Pinz Borges
Doutorado
Universidade Tecnológica Federal do Paraná

Gleifer Vaz Alves
Doutorado
Universidade Tecnológica Federal do Paraná

Augusto Foronda
Doutorado
Universidade Tecnológica Federal do Paraná

Carlos Eduardo Pantoja
Doutorado
Centro Federal de Educação Tecnológica do Rio de Janeiro

PONTA GROSSA

2021

AGRADECIMENTOS

Agradeço principalmente a minha família, em especial meus pais, Pedro Botelho e Lucimar Botelho por todo o suporte que me deram durante a minha graduação. Sem o apoio deles, não seria possível o desenvolvimento deste trabalho.

Agradeço também aos meus orientadores André Pinz Borges e Gleifer Vaz Alves os quais sempre estiveram dispostos e me guiaram desde o inicio deste trabalho. Agradeço por me envolveram com o ambiente de pesquisa, o qual aproveitei muito com a publicação de artigos e participação em eventos. Agradeço muito a paciência de vocês.

Agradeço a todos os meus colegas de faculdade, mas em especial o Othon Briganó, o qual esteve presente comigo desde o inicio da graduação. Agradeço principalmente por toda a companhia durante as aulas e pelas ótimas discussões a respeito de trabalhos.

RESUMO

Sistemas para gerenciamento e alocação de vagas de estacionamentos são desenvolvidos para reduzir tráfego urbano e melhorar a ocupação de estacionamentos. Aplicações desenvolvidas utilizam, de maneira isolada, Sistemas Multi-Agentes (SMA), Sistemas Ciber-Físicos (SCFs), e Computação em Nuvem (CN). Integrar tais tecnologias é fundamental para a criação de aplicativos inteligentes para estacionamentos. Neste trabalho é apresentado a implementação de uma arquitetura que integra as tecnologias de SMA para alocação de vagas, SCFs para detecção de carros e um banco de dados na nuvem para o armazenamento de informações. A integração deu-se por meio de uma API comunicando um sistema de gerenciamento ao dispositivo físico e ao banco de dados, a qual se mostrou eficiente para comunicar os dispositivos.

Palavras-chave: estacionamentos inteligentes; agentes; sistemas ciber-físicos; computação em nuvem.

ABSTRACT

Systems for managing and allocating parking spaces aim to reduce urban traffic and improve the occupancy of parking spaces. Some applications used, in isolated forms, Multi-Agent Systems (MAS), Cyber Physical Systems (CPSs) and Cloud Computing (CC). Integrating these technologies is critical to build smart parking applications. This article presents an implementation of an architecture that integrates MAS technologies for the allocation of vacancies, CPSs for detecting cars and a cloud based database for storing information. The integration took place through a cloud API communicating a management system to the physical device and the database which was proved to be efficient to communicate the devices.

Keywords: smart parking; agents; cyber-physical systems; cloud computing.

LISTA DE ILUSTRAÇÕES

Figura 1 – Ilustração de uma Cidade Inteligente. 19
Figura 2 – Pinagem do NodeMCU 12E 22
Figura 3 – Raspberry Pi Modelo 3B 23
Figura 4 – Sensor ultrassônico modelo HC-SR04 24
Figura 5 – Arquitetura do protocolo MQTT 25
Figura 6 – Arquitetura do protocolo MQTT 25
Figura 7 – Iteração de um agente no ambiente 27
Figura 8 – Estrutura organizacional de um SMA 28
Figura 9 – Exemplo de um agente com comportamento 30
Figura 10 – Diagrama da API 39
Figura 11 – Diagrama de Venn dos trabalhos relacionados 40
Figura 12 – Visão geral da arquitetura 45
Figura 13 – Diagrama de Caso de Uso 48
Figura 14 – Requisição das vagas ao iniciar o sistema 49
Figura 15 – Função que faz a requisição das vagas para a API 50
Figura 16 – Função para requisitar uma vaga no estacionamento 51
Figura 17 – Assinatura da função de requisição de vagas 51
Figura 18 – Função da negociação da vaga 52
Figura 19 – Função de assinatura quando um carro chega na vaga 52
Figura 20 – Função para atualizar a vaga recebida 53
Figura 21 – Diagrama geral do sistema em nuvem 53
Figura 22 – Mapeamento do método *getAllSpots* 55
Figura 23 – Código executado antes da lambda 56
Figura 24 – Exemplo de argumento para o método *getAllSpots* 56
Figura 25 – Operador *switch* na lambda *parking* 57
Figura 26 – Função *getAllSpots* 57
Figura 27 – Exemplo de argumento para o método *requestSpot* 58
Figura 28 – Função *requestSpot* 58
Figura 29 – Tópicos do broker 60
Figura 30 – *Broker* mosquitto em execução 61
Figura 31 – Diagrama do Raspberry Pi 62
Figura 32 – Criação e execução do agente gerente 63
Figura 33 – Criação e execução dos agentes vagas 63
Figura 34 – Vinculação dos comportamentos do agente gerente 63
Figura 35 – Execução da configuração do MQTT 64
Figura 36 – Inscrição do agente gerente nos tópicos das vagas no *broker* 64
Figura 37 – Comportamento *AcknowledgeSpots* 65
Figura 38 – Comportamento *Mqtt* 65
Figura 39 – Configuração de um agente vaga 66
Figura 40 – Comportamento *UpdateSpot* 67
Figura 41 – Método que publica no tópico do *broker* 68
Figura 42 – Método que atualiza uma vaga para ocupado na API 68
Figura 43 – Método que atualiza uma vaga para livre na API 68
Figura 44 – Diagrama do módulo ESP-12e 69
Figura 45 – Função *setup* 70
Figura 46 – Função para o cálculo da distância 71

Figura 47 – Função de *callback* do MQTT 71
Figura 48 – Função para acender os *leds* 72
Figura 49 – Função *loop* 73
Figura 50 – Função de conexão ao *broker* MQTT 74
Figura 51 – Agentes criados no Raspberry 76
Figura 52 – *Broker* com dispositivos conectados 77
Figura 53 – SMA e *broker* MQTT rodando no Raspberry 78
Figura 54 – Duas vagas livres no estacionamento 79
Figura 55 – Estado do banco de dados com duas vagas livres 79
Figura 56 – Tela do Motorista 111.111.111-11 80
Figura 57 – Vaga A1 no estacionamento 80
Figura 58 – Tela do Motorista 222.222.222-22 81
Figura 59 – Vaga A2 no estacionamento 81
Figura 60 – Motorista 222.222.222-22 requisitando vaga A2 82
Figura 61 – Requisição da vaga A2 com sucesso 82
Figura 62 – Requisição da vaga A1 com sucesso 83
Figura 63 – Estado do banco de dados após requisições 83
Figura 64 – Motorista estaciona na vaga A1 84
Figura 65 – Troca de mensagens entre os dispositivos e os agentes 84
Figura 66 – Motorista estaciona na vaga A2 85
Figura 67 – Troca de mensagens entre os dispositivos e os agentes 85
Figura 68 – Tela do administrador do estacionamento 86
Figura 69 – Duas vagas ocupadas no estacionamento 86
Figura 70 – Estado do banco de dados após estacionar 87
Figura 71 – Uma vaga livre no estacionamento 88
Figura 72 – Estado do banco de dados com uma vaga livre 88
Figura 73 – Tela do Motorista 111.111.111-11 89
Figura 74 – Tela do Motorista 222.222.222-22 89
Figura 75 – Vaga A1 no estacionamento 90
Figura 76 – Motorista 111.111.111-11 requisitando vaga A1 90
Figura 77 – Motorista 222.222.222-22 requisitando vaga A1 91
Figura 78 – Estado do banco de dados após requisições 91
Figura 79 – Agentes criados no Raspberry 92

LISTA DE TABELAS

Tabela 1 – Configuração dos cenários do Experimento 3 92
Tabela 2 – Resultados dos cenários do Experimento 3 93

LISTA DE ABREVIATURAS E SIGLAS

ACC	*Agent Communication Channel*
ACL	*Agent Communication Language*
AMS	*Agent Management System*
API	*Application Programming Interface*
AWS	*Amazon Web Services*
DF	*Directory Facilitator*
FaaS	*Function as Service*
FIPA	*Foundation for Intelligence Physical Agents*
IPB	Instituto Politécnico de Bragança
IoT	*Internet of Things*
JADE	*JAva Agent DEvelopment Framework*
MQTT	*Message Queuing Telemetry Transport*
SCF	Sistemas Ciber-Físicos
SMA	Sistemas Multi-Agentes
TCP/IP	*Internet Protocol Suite*
UTFPR	Universidade Tecnológica Federal do Paraná

SUMÁRIO

1 INTRODUÇÃO ... 13
1.1 Objetivos ... 13
1.1.1 Objetivo Geral ... 13
1.1.2 Objetivos Específicos ... 13
1.2 Justificativa ... 13
1.3 Delimitações do Trabalho ... 13
1.4 Contribuições ... 13
1.5 Organização do Trabalho ... 13
2 REFERENCIAL TEÓRICO ... 14
2.1 Estacionamento Inteligente ... 14
2.2 Sistemas Ciber-Físicos ... 20
2.2.1 Módulo ESP8266 ... 21
2.2.2 Rapsberry Pi ... 22
2.2.3 Módulo HC-SR04 ... 23
2.2.4 Protocolo MQTT ... 24
2.3 Agentes e Sistemas Multi-Agentes ... 26
2.3.1 Framework JADE ... 29
2.3.1.1 Comportamentos ... 29
2.3.1.2 Implementação de um agente ... 30
2.3.1.3 Comunicação ... 31
2.3.1.4 Arquitetura ... 32
2.4 Sistemas Web ... 33
2.5 Computação em Nuvem ... 34
2.5.1 Lambda ... 37
2.5.2 DynamoDB ... 38
2.5.3 AppSync ... 38
3 TRABALHOS RELACIONADOS ... 40
3.1 Trabalhos do Grupo de Pesquisa ... 43
3.2 Considerações Finais ... 44
4 DESENVOLVIMENTO DO SISTEMA ... 45
4.1 Arquitetura ... 45
4.2 Sistema de Gerenciamento de Estacionamento ... 47
4.2.1 Usuários ... 47

4.2.2 Funcionamento geral 49
4.3 Computação em nuvem 53
4.3.1 Banco de dados 54
4.3.2 Funções lambdas 55
4.4 Sistema cíber-físico 59
4.4.1 Protocolo MQTT 59
4.4.2 Raspberry Pi e Agentes 61
4.4.3 Módulo ESP-12e 68
4.5 Considerações Finais 74
5 RESULTADOS 75
5.1 Cenários de Teste 75
5.2 Configurações dos Experimentos 76
5.3 Experimento 1 78
5.4 Experimento 2 87
5.5 Experimento 3 92
5.6 Considerações finais 93
6 CONCLUSÃO 94
6.1 Trabalhos Futuros 95
REFERÊNCIAS 96

1 INTRODUÇÃO

O avanço da tecnologia proporcionou meios de facilitar a vida do ser humano. Hoje tem-se uma infinidade de novas tecnologias que se tornaram acessíveis e comuns no século XXI. Este avanço criou possibilidades de novas ideias e conceitos, dentre eles as Cidades e os Estacionamentos Inteligentes. Cidade Inteligente é uma cidade onde tecnologia da informação e comunicação são associadas com infraestruturas usando novas tecnologias (BATTY *et al.*, 2012).

A divergência entre a falta de espaço nas grandes cidades e o número de carros presentes nelas faz com que encontrar vagas seja um problema para os motoristas. Dirigir à procura de uma vaga faz com que o trânsito se torne mais lento, causando congestionamentos, gasto de tempo e gasolina dos veículos e ainda emite no meio ambiente gases poluentes que não seriam emitidos se o carro estivesse estacionado. Estima-se que cerca de um terço dos veículos que circulam em uma cidade a qualquer momento são motoristas enquanto procuram estacionamento, além do fator de frustração, os motoristas que procuram vaga para estacionar criam congestionamento no trânsito (RIZVI *et al.*, 2018).

Atualmente, o método mais comum de se encontrar um lugar para estacionar é procurando uma vaga sem o auxílio de sistemas computacionais. O motorista busca encontrar uma vaga por meio de tentativas, sem utilizar um sistema de informações sobre vagas disponíveis. A solução menos eficiente para esse problema é aumentar o número de vagas, assim os motoristas terão maiores possibilidades de encontrar um estacionamento. Contudo, há métodos inteligentes e, consequentemente, mais eficientes para atenuar ou até resolver o problema, como os estacionamentos inteligentes (GENG *et al.*, 2013).

Estacionamento Inteligente é um sistema de estacionamento que auxilia os motoristas a encontrar vagas utilizando sensores que detectam se há ou não um veículo e então o direcionam para a vaga (HASSOUNE *et al.*, 2016). O objetivo desses sistemas é garantir que o motorista encontre uma vaga para estacionar da maneira mais rápida possível levando em consideração o local desejado. Assim, o motorista pode reservar uma vaga antes mesmo de entrar no veículo (CASTRO *et al.*, 2017).

Desse modo, com o número da vaga já de posse do motorista, é necessário que o estacionamento disponha de uma arquitetura física para realizar o controle da

vaga e dos setores do estacionamento, como detectar quando um carro é estacionado. Ainda, é necessário que o estacionamento possua um meio do administrador visualizar o estado do estacionamento e dos motoristas requisitarem as vagas. Assim, por ser uma solução complexa, estacionamentos inteligentes, por serem ambientes dinâmicos, precisam adotar tecnologias que atendam a demanda da implementação e que facilite na representação do ambiente. Uma das alternativas dá-se através da integração de diferentes tecnologias como agentes inteligentes, Sistemas Ciber-Físicos (SCFs) e Computação em Nuvem (CN).

Um agente é um sistema computacional capaz de ações autônomas no ambiente que está situado visando atingir seus objetivos propostos (WOOLDRIDGE, 2009), SCFs são elementos computacionais que oferecem comunicação com entidades físicas utilizando plataformas de hardware (KAITHAN *et al.*, 2015) e a CN é um termo para a disponibilidade sob demanda de recursos do computacionais. Desta forma, desenvolver agentes e embarcá-los em plataformas de hardware atualizando constantemente um sistema em nuvem, é uma alternativa considerável para fazer a conexão do motorista em busca da vaga com o sistema real de estacionamentos inteligentes.

Assim, este trabalho propõe a criação de uma aplicação que permita motoristas requisitarem vagas e administradores visualizarem, em tempo real, as condições do estacionamento. Um protótipo de uma arquitetura ciber-física com o intuito de obter informações sobre a vaga para que, a partir destas informações, seja possível atualizar o estado do estacionamento. E uma solução em nuvem para integrar esses dois componentes, controlando as vagas do estacionamento e as requisições realizadas pelos usuários e pelo sistema ciber-físico.

A aplicação para gerenciamento do estacionamento foi desenvolvida utilizando, a biblioteca Javascript, React para permitir que motoristas e administradores tenham acesso à visualização das vagas do estacionamento. A solução em nuvem foi implementada através da Amazon Web Services (AWS) para controlar as requisições, integrando os motoristas e administradores ao sistema físico. E a arquitetura física é constituída de placas Raspberry Pi contendo agentes embarcados e implementados no *framework* JADE para receber as informações da parte física, e módulos ESP-12e conectados a sensores de presença para monitorar o estado das vagas (por exemplo, livre ou ocupada).

O Java Agent DEvelopment Framework (JADE) é um *framework* para o de-

senvolvimento de agentes totalmente implementado em Java (GREENWOOD *et al.*, 2004), o qual foi utilizado como linguagem de implementação do agente pois é possível executar uma aplicação Java no Raspberry Pi (RASPBERRY, 2019). A comunicação entre o Raspberry e os módulos ESP-12e foi dada por meio do protocolo Message Queuing Telemetry Transport (MQTT), utilizado para troca de mensagens que utiliza a camada TCP/IP e é ideal para conexões de máquina para máquina (MQTT, 2019).

O protótipo desenvolvido tem a função de gerenciar um estacionamento inteligente por meio da integração de componentes. A verificação da aplicação será realizada explorando situações onde o motorista estaciona seu veículo em uma vaga e o administrador do estacionamento consegue verificar este acontecimento em tempo real. Através dessa integração, foi permitido aos motoristas estacionarem seus veículos tendo sua vaga previamente alocada. Do mesmo modo, a comunicação entre os componentes físicos e em nuvem permitiu que o administrador pudesse ter controle do estacionamento.

1.1 Objetivos

Nesta seção serão abordados os objetivos geral e específicos a serem atingidos pelo presente trabalho.

1.1.1 Objetivo Geral

O objetivo geral deste trabalho é desenvolver um protótipo de um sistema ciber-físico baseado em agentes inteligentes utilizando plataformas de hardware e integrar com um sistema de gerenciamento de um estacionamento inteligente utilizando a computação em nuvem.

1.1.2 Objetivos Específicos

Para atingir o objetivo geral deste trabalho é necessário atingir os seguintes objetivos específicos:

- Implementar um protótipo físico de simulação de vagas usando ESP-12e e hardwares necessários;
- Implementar um agente responsável por receber as informações da parte física e alocar a vaga;

- Embarcar o agente implementado no Raspberry Pi;
- Implementar a comunicação do Rasperry Pi com o ESP-12e via protocolo MQTT;
- Implementar uma API com comunicação com um banco de dados para armazenar as informações do estacionamento;
- Hospedar a API e o banco de dados na *Amazon Web Services* (AWS);
- Implementar um sistema de gerenciamento de estacionamento capaz de consumir a API desenvolvida;
- Integrar o protótipo físico com o sistema de gerenciamento de estacionamento desenvolvido;
- Realizar experimentos e analisar os resultados obtidos.

1.2 Justificativa

(DI NAPOLI *et al.*, 2014b) concluíram em sua pesquisa que aplicações de estacionamentos inteligentes deverão ajudar o motorista a encontrar uma vaga de estacionamento e, como consequência, ajudar a reduzir congestionamentos e também reduzir emissão de carbono na atmosfera. Uma pesquisa realizada mostra que um sistema de estacionamento inteligente implantado economizou 30,56km de 633 motoristas, assim como uma economia de 64,3kg de emissão de CO2 no meio ambiente em uma simulação realizada em um período de duas semanas (LIU *et al.*, 2014).

No projeto Smart Parking desenvolvido em parceria pelo IPB com a UTFPR-PG estão sendo elaborados protocolos de comunicação e negociação entre agentes (CASTRO *et al.*, 2017) (DUCHEIKO *et al.*, 2018). O projeto, além da negociação, aborda também a implementação de um estacionamento inteligente como um todo, contendo sistemas físicos (BOTELHO, P. *et al.*, 2019) (SAKURADA *et al.*, 2019). Porém, o projeto ainda não conta com uma aplicação que o usuário possa utilizar para fazer a reserva de vagas e também não conta com um meio de integrar todos estes componentes, sendo estes itens propostos aqui.

Os trabalhos desenvolvidos no projeto até o momento, realizam simulações computacionais para testar sua aplicabilidade em um estacionamento inteligente. A integração dos agentes com o hardware possibilitará, em trabalhos futuros, a realização de diferentes testes em partes reais. Assim como realizar os testes dos trabalhos já

desenvolvidos em um ambiente mais próximo de um estacionamento real.

As tecnologias para o desenvolvimento do protótipo ciber-físico foram escolhidas principalmente pelo baixo custo de instalação, visto que as placas de maior custo são utilizadas apenas sob demanda do estacionamento, ou seja, quanto maior o estacionamento, mais placas será necessário. O JADE é um *framework* para o desenvolvimento de agentes implementado na linguagem Java, como o Raspberry é um microprocessador, é possível executar uma aplicação Java nele, portanto, os agentes embarcados farão o trabalho de maneira autônoma, sem a necessidade de intervenção humana. A AWS foi utilizada sobretudo pelo fácil uso, pela confiabilidade e pela segurança. Uma vantagem do uso da AWS é o ambiente gratuito, já que os serviços são cobrados a partir de um número alto de utilização.

1.3 Delimitações do trabalho

Esta seção descreve as delimitações e escopo do presente trabalho.

- Foi desenvolvido um protótipo ciber-físico com apenas duas vagas físicas devido a limitação de recursos de hardware, as demais vagas são representadas de forma virtual;
- O protótipo ciber-físico consiste em um Raspberry com um SMA embarcado. Dois ESP-12e para controle das vagas, dois sensores ultrassônicos para detecção de veículos e 4 *leds* para indicar se a vaga está livre ou ocupada.
- Um SMA em JADE foi desenvolvido para gerenciar a parte física do estacionamento. Há um agente gerente responsável por um setor do estacionamento e outros *N* agentes vagas, sendo dois desses *N*, responsáveis pelas duas vagas físicas do protótipo;
- O sistema em nuvem foi desenvolvido no ambiente gratuito da AWS, o qual possui uma quantidade limitada de requisições mensais. Para o banco de dados há um limite de 25GB de armazenamento e para a API há um limite de 250 mil requisições gratuitas por mês.
- Foi desenvolvido um sistema de gerenciamento para que fosse possível ilustrar em tempo real todas as vagas e quais estão livres e ocupadas. Neste sistema também é possível requisitar vagas;
- Todos os testes e simulações apresentados neste trabalho não abordam

características como a negociação das vagas e o abrir e fechar de cancelas do estacionamento;

- A negociação das vagas é realizada de maneira aleatória para os motoristas, visto que, outros trabalhos do grupo de pesquisa já lidam com a abordagem de negociação de vagas;
- Neste trabalho, considera-se que os motoristas obedecem rigorosamente as regras e estacionam somente na vaga requisitada e obtida pelo motorista.

1.4 Contribuições

Nesta seção descrevem-se as contribuições almejadas por este trabalho.

- Integração de três elementos distintos, um sistema ciber-físico responsável por detectar o estado físico da vaga, um sistema em nuvem responsável por controlar as requisições pelo armazenamento das informações da vaga e um sistema web que permite a visualização das vagas pelo administrador e a requisição de vagas pelo motoristas. A integração destes elementos permitem a comunicação em tempo real dos componentes da arquitetura, mitigando chances de falha do sistema;
- Protótipo ciber-físico funcional e modular que pode ser facilmente implantado e de baixo custo;
- Ambiente em nuvem hospedado na AWS e de baixo custo, oferecendo confiabilidade e segurança além de oferecer os serviços com escalabilidade automática não afetando a performance mesmo quando há um aumento nas cargas de trabalho;
- Avaliação da quantidade máxima de vagas que podem ser alocadas e gerenciadas por um único Raspberry.

1.5 Organização do trabalho

Este trabalho está estruturado em seis capítulos. O capítulo dois apresenta as definições sobre estacionamentos inteligentes, sistemas ciber-físicos, agentes, computação em nuvem. O capítulo três apresenta os trabalhos relacionados. No capítulo quatro, o desenvolvimento do protótipo ciber-físico, do sistemas de gerenciamento e a comunicação

da arquitetura é apresentada. Na sequência, o capítulo cinco apresenta os resultados obtidos. E o capítulo seis apresenta a conclusão do trabalho com os trabalhos futuros.

2 REFERENCIAL TEÓRICO

Este capítulo aborda conceitos fundamentais para a compreensão do presente trabalho, como o conceito de Cidades e Estacionamentos Inteligentes, Sistemas Ciber-físicos, Agentes, Sistemas Multi-Agentes e Computação em Nuvem. Ainda, os trabalhos relacionados são destacados.

2.1 Estacionamento Inteligente

Para compreender o que é um Estacionamento Inteligente (EI), primeiro, necessita-se compreender de onde surgiu esta ideia. Um Estacionamento Inteligente faz parte de uma cidade onde tudo está conectado por meio da tecnologia, a Cidade Inteligente. No século 20, a ideia de uma Cidade Inteligente era, como na mídia, uma ficção científica. Porém, com o avanço da tecnologia e o surgimento de dispositivos computacionais de grande e pequena escala e a possibilidade de uma inteligência ser embarcada nestes dispositivos, a Cidade Inteligente está caminhando para se tornar uma realidade (BATTY *et al.*, 2012).

Em resumo, as Cidades Inteligentes usam tecnologias de comunicação e informação como sensores para coletar dados e então, usa-se estes dados para gerenciar seus recursos de forma eficiente (MCLAREN *et al.*, 2018). Isso inclui coleta de dados de sistemas de tráfego e transporte, usinas elétricas, redes de abastecimento de água, gerenciamento de resíduos, detecção de crime, escolas, bibliotecas, hospitais, dentre vários outros recursos que uma cidade possui (MCLAREN *et al.*, 2018). A Figura 1 ilustra uma Cidade Inteligente.

Figura 1 – Ilustração de uma Cidade Inteligente.

Fonte: (ARTUSSE, 2019)

Para gerenciar os dados de tráfego de maneira eficiente, deve-se, primeiro, resolver um dos maiores problemas causadores de engarrafamento nas cidades, a busca por vagas de estacionamento. Os Estacionamentos Inteligentes são considerados cruciais para ajudar tanto na redução da emissão gases em cidade, quanto na vida dos motoristas que buscam estacionar, fazendo com que estacionar seja mais fácil (DI NAPOLI *et al.*, 2014b).

Os Estacionamento Inteligentes podem ser categorizados em diversos sistemas, como sistema de orientação de estacionamento e informação, sistema de informação com base no trânsito, sistema de pagamento inteligente e estacionamento automatizado (CHINRUNGRUENG *et al.*, 2007). Cada sistema utiliza de diferentes tecnologias para detectar se um carro está ou não estacionado na vaga. (MIMBELA *et al.*, 2000) dividiu os sensores do veículo e o sistema detector em duas categorias, sensores intrusivos e não-intrusivos.

- **Sensores intrusivos:** Sensores que são instalados em buracos na superfície da vaga, por tunelamento sob a estrada. Exemplos de sensores intrusivos são sensores infravermelhos, magnetômetros e sensores resistivos de magneto.
- **Sensores não-intrusivos:** Sensores que podem ser facilmente instalados e mantidos e não afetam a superfície do processo. Sensores não intrusivos abrangem sensores acústicos, sensor infravermelho, RFID, ultra-som e processamento de imagens de vídeo.

Basicamente, em um Estacionamento Inteligente há um sistema que coleta informações da vaga, onde pode-se levar em consideração o tipo da vaga, o local da

vaga, o horário e custo da vaga (RIZVI *et al.*, 2018). Há também sensores que detectam se a vaga está livre ou ocupada, enviando estas informações para o sistema de coleta de dados. Por fim, há um canal de comunicação entre o sistema e o motorista, para que uma negociação da vaga possa ser iniciada.

2.2 Sistemas Ciber-Físicos

Os sistemas de computação modernos apresentam uma combinação de sistemas físicos e elementos computacionais, que são desenvolvidos de forma independente. Porém, as recentes tendências do aumento do desempenho e o uso de padrões de uso complexo mudaram a integração destes componentes de forma significativa. Essas mudanças necessitam de novas abordagens onde componentes físicos e computacionais estão integrados em diversos níveis, assim, essa necessidade levou a avanços no campo de pesquisa de Sistemas Ciber-Físicos (KAITHAN *et al.*, 2015).

SCFs são definidos como sistemas que oferecem integração de computação, redes e componentes físicos. São sistemas onde componentes físicos e computacionais estão profundamente interligados cada um operando em diferentes escalas, interagindo uma com as outras em uma infinidade de maneiras que mudam com o contexto (JAZDI, 2014).

Algumas das características que definem SCF são, capacidade computacional em todos os elementos físicos, alto grau de automação, redes em múltiplas escalas, integração em múltiplas escalas temporais e espaciais e reorganização de dinâmicas (SANISLAV, 2012). A geração anterior à dos SCF é conhecida como sistemas embarcados e encontram aplicação em diversas áreas, como aeroespacial, automotiva, energia, infraestrutura, etc. (KAITHAN *et al.*, 2015). No entanto, sistemas embarcados têm como foco os elementos computacionais, enquanto que SCF dão um destaque maior na ligação entre elementos computacionais e físicos.

Devido a comunicação entre o mundo físico e o mundo virtual ser muito próxima, existem alguns desafios na criação de um SCF. Para uma integração sem falhas, os eventos no mundo físico devem ser refletidos no mundo virtual, e as decisões tomadas no mundo virtual devem comunicar o mundo físico. Ambas tarefas devem ser executadas de maneira precisa e no tempo certo. Para isso, os SCFs precisam ser coordenados por dispositivos de computação e sensores e atuadores distribuídos (KAITHAN *et al.*,

2015).

As próximas sub-seções irão abordar alguns dispositivos computacionais existentes que podem atuar na comunicação entre o mundo físico e o mundo virtual e sobre um protocolo de comunicação bastante utilizado na comunicação entre dispositivos físicos.

2.2.1 ESP8266

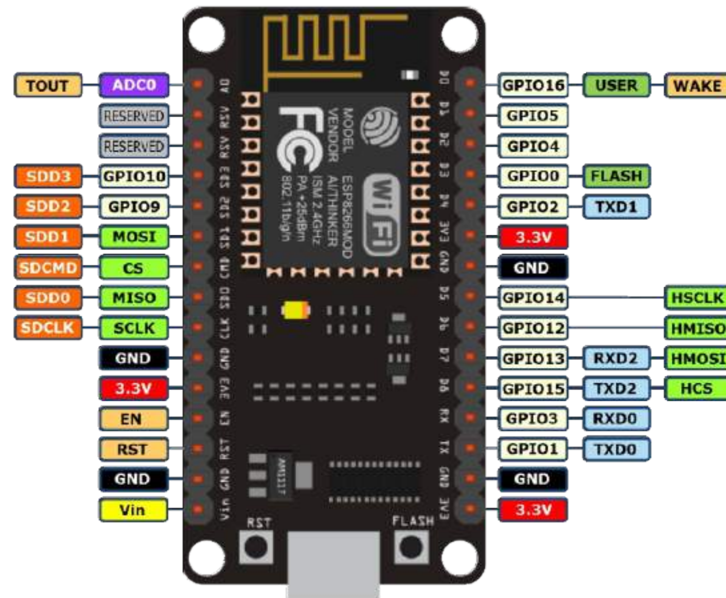
O ESP8266 é um microcontrolador produzido pelo fabricante chinês *Espressif Systems* que inclui capacidade de comunicação por *Wi-Fi*, que é o grande diferencial dele para os outros microcontroladores (SYSTEMS, 2019). Sua antena *Wi-Fi* interna com suporte a conexão *wireless* padrão IEEE 802.11, possibilita o envio de dados coletados para um outro dispositivo sem a necessidade de utilizar cabos para comunicação. Existem diversos tipos de modelos do ESP8266, do ESP-1 ate o ESP-14 e o mais recente produzido, ESP32, que também possui bluetooth 4.0 integrado (SYSTEMS, 2019).

Segundo (ŠKRABA *et al.*, 2017), o ESP8266 pode funcionar em duas configurações, o Access Point e o *client*:

- **Access Point:** O ESP8266 cria um servidor com IP definido ou aleatório e funciona basicamente como um roteador criando uma rede Wi-Fi restrita por login e senha;
- **Client:** É estabelecida uma conexão com a rede Wi-Fi fornecida e todos os dispositivos conectados na mesma rede possuem acesso ao ESP8266 pelo endereço de IP.

O módulo Node MicroController Unit (NodeMCU) é uma solução encapsulada para *Internet of Things* (IoT) que possui um ESP8266 embutido que pode ser programado nativamente na linguagem LUA, porém existem frameworks que permitem a programação do mesmo utilizando a IDE do Arduino (GROKHOTKOV, 2019). Este módulo facilita o desenvolvimento de aplicações que utilizam os microcontroladores da família ESP, já que possui o mesmo soldado na placa e todos os circuitos necessários para seu correto funcionamento. A Figura 2 ilustra a configuração de pinos da placa modelo 12E.

Figura 2 – Pinagem do NodeMCU 12E



Fonte: (ARDUINING, 2019)

Este dispositivo foi projetado de forma modular, ou seja, cada sensor é independente e pode ser adicionado ou removido sem falhas. Pode ser verificado na Figura 2 que o NodeMCU possui uma entrada analógica e 13 pinos digitais, o que é suficiente para conectar um sensor de presença e dois LED's, por exemplo. Uma grande vantagem de utilização deste microcontrolador é o baixo custo e o alcance do *Wi-Fi* de até 120m em local aberto.

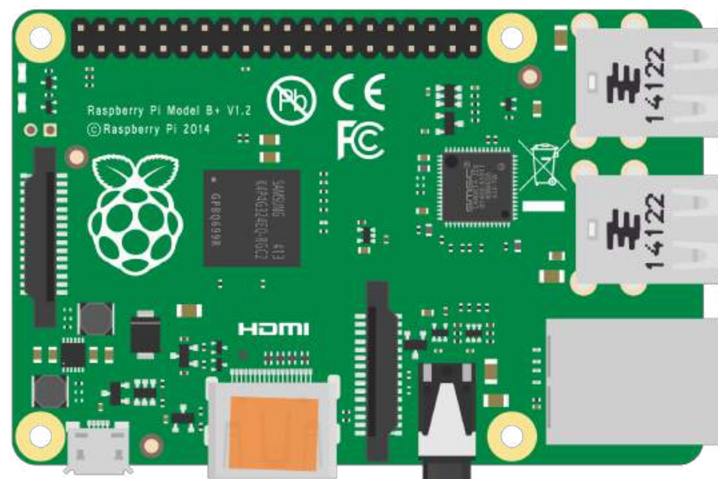
2.2.2 Raspberry Pi

O Raspberry Pi é um computador do tamanho de um cartão de crédito desenvolvido no Reino Unido pela fundação Raspberry Pi (SJOGELID, 2013). O Raspberry Pi possui mais poder computacional, mais memória e maior capacidade de armazenamento do que os primeiros computadores pessoais, porém, não possui o desempenho e a capacidade de disco rígido dos computadores atuais. As limitações do Raspberry Pi podem ser superadas se conectada a periféricos como os próprios discos rígidos, monitores HDMI e até sistemas de som de alta qualidade (UPTON *et al.*, 2013).

Os primeiros modelos do Raspberry Pi foram anunciados em 2011 e lançados em 2012, idealizados pelo inglês Pete Lomas para serem os computadores mais baratos do mundo, com preços a partir de 25 dólares (Modelo A) e 35 dólares (Modelo B). Foram criados para ter finalidades educativas, estimulando o ensino da Ciência da Computação e áreas relacionadas (RASPBERRY, 2019). Há diversos projetos sendo

desenvolvidos utilizando-a, alguns exemplos são: automação e segurança residencial, central multimídia, monitoramento de clima, controlador de robôs e drones e até a utilização do mesmo como um servidor (UPTON *et al.*, 2013). A Figura 3 ilustra a pinagem e as entradas e saídas do Raspberry Pi.

Figura 3 – Raspberry Pi Modelo 3B



Fonte: (RASPBERRY, 2019)

Como qualquer computador, o Raspberry precisa de um sistema operacional, assim, o *Raspbian* [1] é o sistema operacional padrão do Raspberry Pi. O *Raspbian* deriva da distribuição Linux Debian, permitindo, por ser Linux, a instalação de pacotes e dependências via linha de comando (HARRINGTON, 2015).

2.2.3 Módulo HC-SR04

O módulo HC-SR04 é um sensor com o qual é possível medir a distância de um objeto que se encontra em sua frente. Um pulso eletrônico é emitido pelo dispositivo e o mesmo se reflete no objeto a sua frente e retorna para o sonar, assim, com base no tempo entre a emissão do pulso e a recepção, é possível determinar a distância do objeto (TECHNOLOGIES, 2019). A Figura 4 ilustra o sensor e seus pinos.

[1] https://www.raspbian.org.

Figura 4 – Sensor ultrassônico modelo HC-SR04

Fonte: (TECHNOLOGIES, 2019)

O pulso emitido pelo sensor é feito de forma automática e no intervalo de tempo definido pelo programador. A frequência do pulso emitida é de 40KHz, a equação 1 mostra como é obtida a velocidade (TECHNOLOGIES, 2019).

$$\Delta S = \frac{HLT * vSom}{2} \tag{1}$$

Onde S é a distância, HLT é o tempo que o pulso demorou para ir e voltar e $vSom$ é a velocidade do som, considerada igual a 340m/s, assim, a distância obtida será em m/s. A divisão por 2 se deve ao fato que a onda é enviada e recebida, percorrendo duas vezes a distância procurada.

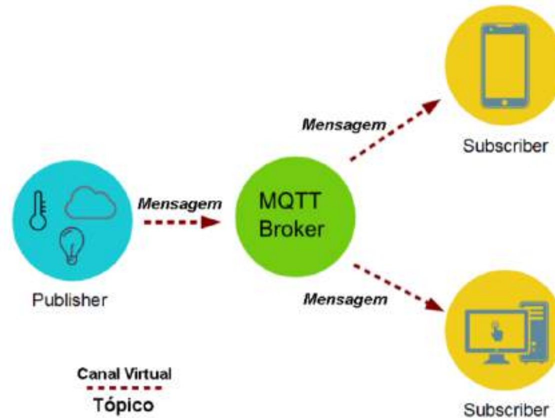
2.2.4 Protocolo MQTT

Para que dispositivos se conectem à Internet é necessário que sejam definidos modelos e padrões que atendam suas reais necessidades. Em muitos casos, estes dispositivos são limitados em recursos, tais como memória, processamento e fornecimento de energia. Portanto, enfatizam a utilização de protocolos de comunicação simples e de fácil implementação, sem que haja um forte apelo para a segurança, devido sua simplicidade.

O MQTT é um protocolo leve para comunicação que funciona em cima do Protocolo de Controle de Transmissão, tendo a porta 1883 definida por padrão para comunicação do servidor com o cliente pela Autoridade de Atribuição de Números da Internet (MQTT, 2019). Seu funcionamento é baseado no princípio de publicação de mensagens em um tópico e na leitura de tópicos. Desta forma, sempre que um dispositivo for se comunicar com outro, deve-se especificar qual o tópico para qual a informação será enviada, da mesma forma, a aplicação que deseja receber esta

informação deve estar inscrita neste tópico. A Figura 5 ilustra a arquitetura básica do protocolo MQTT.

Figura 5 – Arquitetura do protocolo MQTT



Fonte: Adaptado de (AZZOLA, 2019)

Para que haja esta comunicação é necessário fazer o uso de um *broker*, que atua como um servidor no qual os dispositivos se conectam para se publicar e se inscrever nos tópicos. A Figura 6 ilustra como é feita a conexão de uma aplicação em um *broker* na linguagem Java.

Figura 6 – Arquitetura do protocolo MQTT

```
1  import org.eclipse.paho.client.mqttv3.MqttClient;
2  import org.eclipse.paho.client.mqttv3.MqttException;
3  import org.eclipse.paho.client.mqttv3.MqttMessage;
4  import classes.MqttCallback;
5
6  public class MqttExample {
7      public static MqttClient client;
8      public static MqttMessage message = new MqttMessage();
9      static String broker = "tcp://192.168.0.108:1883";
10     static String topic = "estacionamento/setor1/vaga23"
11
12     client = new MqttClient(broker, "MqttClientName");
13     client.setCallback(new MqttCallback());
14     client.connect();
15
16     client.subscribe(topic);
17     client.publish(topic, "Just subscribed");
18 }
```

Fonte: Autoria própria (2021)

Neste exemplo, a biblioteca do Eclipse Mosquitto é importada, na linha 9 é definido o endereço IP do *broker* e na linha 12 a conexão com o *broker* é instanciada. Na linha 13 é setado um *callback*, que é a função que irá executar assim que uma

mensagem no tópico inscrito chegar. Na linha 14 a conexão com o *broker* é criada e na linha 16 mostra como é feita inscrição em um tópico e na linha 17 como é feita uma publicação em um tópico.

Não há necessidade de criar estes tópicos, porém, para haver uma comunicação sem falhas, o mesmo nome de tópico deve ser programado tanto para publicação, quanto para inscrição. Uma maneira de padronizar os tópicos é criando níveis de hierarquia, por exemplo, um estacionamento que possui vários setores e cada setor possui várias vagas a padronização pode seguir a seguinte forma: estacionamento/setor1/vaga23 como foi mostrado na Figura 6.

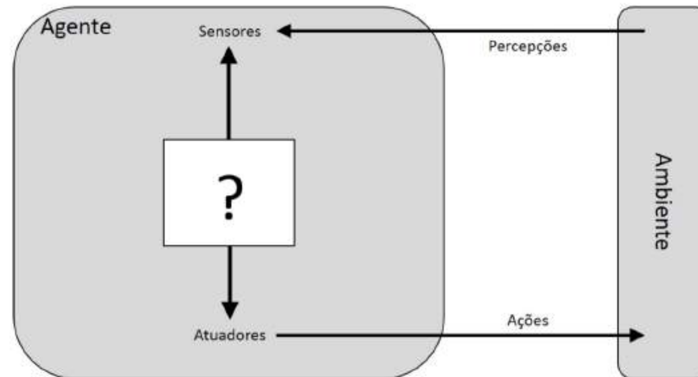
Assim, um dispositivo que quiser saber se a vaga 23 está livre ou ocupada irá se inscrever neste tópico, recebendo assim qualquer publicação. Da mesma maneira, o dispositivo responsável pela vaga 23 deve publicar neste mesmo tópico sempre que houver uma mudança de estado na vaga.

2.3 Agentes e Sistemas Multi-Agentes

Um agente é um software que pode ser visto com as capacidades de percepção de seu ambiente por via de sensores e age nesse ambiente por meio de atuadores. O termo percepção é usado para se referir às entradas perceptivas do agente em qualquer instante. A sequência de percepção de um agente é um histórico de tudo o que o agente já percebeu no ambiente situado. Assim, permite-se que o agente consiga atuar no ambiente tomando ações a partir de decisões tomadas com base em seu objetivo (RUSSELL *et al.*, 1995).

A Figura 7 ilustra a estrutura básica de um agente. O agente recebe informações através de seus sensores, executa um raciocínio definido e produz como saída uma ação que afeta o ambiente. Fazendo uma analogia a nós, seres humanos, nossos sensores são olhos e ouvidos e nossos atuadores são pernas e mãos. Um agente robótico pode ter câmeras e rastreadores de infravermelho para sensores e motores para atuadores.

Figura 7 – Iteração de um agente no ambiente



Fonte: (RUSSELL *et al.*, 1995)

Existem diferentes definições do que é um agente, porém, existe um consenso que os agentes possuem quatro características principais: autonomia, proatividade, reatividade e interatividade social (WOOLDRIDGE, 2009).

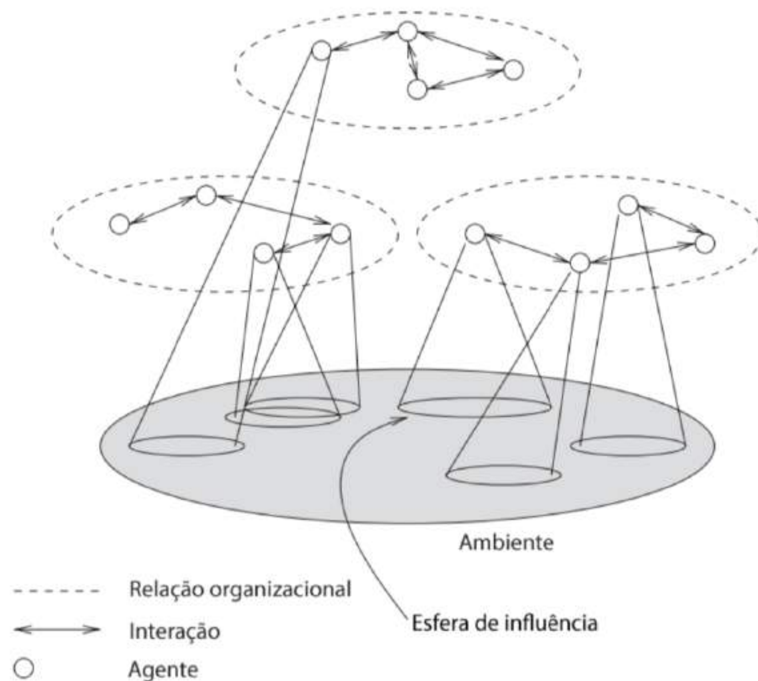
- **Autonomia:** É o poder de decidir quais ações tomar visando sempre atingir um objetivo. Diferentemente de outras técnicas de programação, onde é seguido uma sequência de regras pré-definidas para realizar uma tarefa, no contexto de agentes, isso não é aplicado, pois o comportamento de um agente é autônomo de maneira que pode-se tomar decisões diferentes em uma mesma situação;
- **Proatividade:** É a capacidade de exibir um comportamento direcionado por objetivos tomando uma iniciativa para satisfazê-los;
- **Reatividade:** É a capacidade de perceber o ambiente e tomar ações ações de modo a satisfazer seus objetivos. Estas ações podem ser racionais, baseadas em conhecimentos adquiridos previamente ou reflexivas, uma ação gerada a partir de um impulso vindo do ambiente; e
- **Habilidade Social:** É a capacidade de um agente interagir com outros agentes ou com seres humanos para atingir seus objetivos. Esta característica é definida pela habilidade de colaboração com outros agentes, comunicação por meio da troca de informações e a capacidade de chegar-se a um acordo com outros agentes. Comumente, estes sistemas compostos são chamados de Sistemas Multi-Agentes.

A dificuldade para encontrar uma definição se deve ao fato que o conceito de agentes muda dependendo do campo de aplicação (WOOLDRIDGE, 2009). Por

exemplo, em algumas áreas, a capacidade de um agente aprender pode não ser importante, enquanto que em outras esta mesma capacidade é de suma importância.

Com base na estrutura ilustrada na Figura 7, é possível estabelecer que um agente pode ser distribuído em vários agentes, cooperando um com os outros de forma a solucionar problemas complexos. Esta cooperação se dá por meio de suas habilidades sociais e não necessariamente é relacionada ao compartilhamento de recursos. Sistemas com essa característica são chamados de Sistemas Multi-Agentes (SMAs). A Figura 8 ilustra a organização de um SMA.

Figura 8 – Estrutura organizacional de um SMA



Fonte: Adaptado de (BORDINI *et al.*, 2007)

Um SMA é um conjunto de agentes que compartilham o mesmo ambiente e possuem maneiras de se comunicarem entre si. Dentro deste sistema, um agente pode ter uma hierarquia diferente de outro, tendo assim um grau maior ou menor de influência dentro do ambiente. Cada agente em um SMA possui uma capacidade de percepção do ambiente diferente do outro, influenciando assim diferentes aspectos do ambiente (WOOLDRIDGE, 2009). A próxima sub-seção irá tratar do *framework* para a programação de agentes utilizado neste trabalho, o JADE.

2.3.1 *Framework* JADE

O JADE é um *framework* para programação de agentes totalmente implementado na linguagem Java, o que pode ser uma grande vantagem devido ao enorme conjunto de recursos e bibliotecas oferecidas pela linguagem (JADE, 2003). Com o objetivo de facilitar o desenvolvimento de aplicações baseados em agentes por meio de um ambiente, O JADE implementa os recursos de suporte de ciclo de vida exigidos pelos agentes, a lógica central dos agentes e um conjunto avançado de ferramentas gráficas (GREENWOOD *et al.*, 2004).

Como este *framework* foi totalmente desenvolvido em linguagem Java, o JADE oferece um grande conjunto de abstrações que permite aos desenvolvedores criar SMA com mínima experiência em agentes. Outro fator para a escolha da linguagem Java é de que linguagens orientadas a objetos são consideradas adequadas para desenvolvimento de agentes, pelo fato de o conceito de agentes não estar distante do conceito de objetos (GREENWOOD *et al.*, 2004).

2.3.1.1 Comportamentos

O JADE disponibiliza aos usuários funcionalidades para implementação dos comportamentos do agente a ser desenvolvido. Estes comportamentos apresentam dois métodos, o método *action* que define as tarefas a serem executadas quando determinado comportamento for acionado, e o método *done* que retorna um valor do tipo *boolean* quando finalizada a execução de um comportamento (GREENWOOD *et al.*, 2004).

Um agente pode executar vários comportamentos de forma simultânea, porém estes comportamentos não são executados todos ao mesmo tempo, como *threads* nas linguagens de programação, mas de forma sequencial. Assim, quando um comportamento é acionado seu método *action* é executado até que sua execução seja finalizada. É responsabilidade do desenvolvedor definir quando cada comportamento será executado (GREENWOOD *et al.*, 2004).

Os comportamentos podem ser de três tipos:

- ***One-Shot*:** tem como finalidade ser completado em apenas uma execução, sendo a função *action* executada apenas uma vez. Para utilizar este comportamento é necessário estender a classe *OneShotBehaviour*;

- ***Cyclic***: tem como finalidade nunca ser completado, sendo a função *action* constantemente chamada até que a execução do agente seja finalizada. Para utilizar este comportamento é necessário estender a classe *CyclicBehaviour*;
- ***Generic***: tem como finalidade executar diferentes tarefas de acordo com um valor de status. Sua execução é concluída quando determinada condição é atendida. Para utilizar este comportamento é necessário estender a classe *Behaviour*.

2.3.1.2 Implementação de um agente

Para utilizar o JADE no desenvolvimento de um agente, basta adicionar a biblioteca do JADE no projeto e importar as classes necessárias. Para a criação de um agente, é necessário estender a classe do tipo *Agent* e sobrescrever a função *setup*. Também, é necessário adicionar os comportamentos que o agente deve adotar. A Figura 3 mostra um exemplo de como implementar um agente simples com um comportamento qualquer utilizando JADE.

Figura 9 – Exemplo de um agente com comportamento

```
1  import behaviours.MyBehaviour;
2  import jade.core.Agent;
3
4  public class MyAgent extends Agent {
5      @Override
6      protected void setup() {
7          System.out.println("Agent name " + getAID().getName());
8      }
9
10     addBehaviour(new MyBehaviour());
11 }
```

Fonte: Autoria própria (2021)

Nas linhas 1 e 2 são feitas as importações. Na linha 4, a classe *MyAgent* estende a classe *Agent* da biblioteca do JADE. Na linha 6, a função *setup* padrão da classe *Agent* é sobrescrita e então são inseridas todas as instruções do agente, neste exemplo, apenas o nome do agente é impresso na tela. Na linha 10, um *OneShotBehaviour* é atrelado ao agente.

2.3.1.3 Comunicação

A *The Foundation for Intelligent Physical Agents* (FIPA) é uma organização internacional de padrões sem fins lucrativos que promove a tecnologia baseada em agentes e a interoperabilidade de seus padrões com outras tecnologias. Foi formada para produzir especificações de padrões de software para agentes e assim promover o sucesso no desenvolvimento de aplicações, serviços e equipamentos baseados em agentes (FIPA, 2021).

Dentro da FIPA, é especificada a linguagem de comunicação entre agentes, denominada de Agent Communications Language (FIPA-ACL). Esta linguagem contém um conjunto de parâmetros para uma efetiva comunicação entre agentes. Os parâmetros necessários para realização de uma troca de mensagens se dão de acordo com a situação, sendo obrigatórios em qualquer mensagem os parâmetros de performativa, agente remetente, agente destinatário e conteúdo.

A linguagem FIPA-ACL também define o tipo da mensagem a ser enviada, indicando ao agente destinatário a intenção da troca de mensagens por parte do agente remetente (GREENWOOD *et al.*, 2004). Cada uma dessas ações define um tipo de resultado esperado, por exemplo, uma mensagem do tipo *REQUEST*, utilizada nesse trabalho, que o agente remetente está solicitando uma ação para ser executada pelo agente destinatário. As demais ações são definidas por:

- *AGREE:* Ação de concordar em executar certa tarefa;
- *CANCEL:* Ação em que o agente remetente informa ao agente destinatário que não há mais intenção do destinatário executar determinada tarefa;
- *CALL FOR PROPOSAL:* Ação de solicitar propostas para executar determinada tarefa;
- *CONFIRM:* Ação em que o agente remetente informa ao agente destinatário que determinada proposição é verdadeira;
- *INFORM:* Ação em que o agente remetente informa ao agente destinatário que determinada proposição é verdadeira;
- *PROPOSE:* Ação de enviar uma proposta para execução de determinada tarefa, considerando condições prévias;
- *REQUEST:* Ação em que o agente remetente solicita a execução de determinada tarefa ao agente destinatário.

2.3.1.4 Arquitetura

A arquitetura de comunicação JADE tenta oferecer mensagens flexíveis e eficientes, escolhendo de forma transparente o melhor transporte disponível e aproveitando uma tecnologia distribuída e incorporada no ambiente de tempo de execução Java. Embora apareça como uma entidade única para o mundo externo, a plataforma do JADE é em si um sistema distribuído, uma vez que pode ser dividida em vários *hosts*. O sistema JADE compreende um ou mais contêineres de agente, cada um vivendo em uma máquina virtual Java separada e fornecendo suporte de ambiente de execução para alguns agentes JADE (BELLIFEMINE *et al.*, 2001).

O Java RMI (*Remote Method Invocation*) é usado para comunicação entre os contêineres e cada um deles também pode atuar como um cliente IIOP (*Internet Inter-ORB Protocol*) para encaminhar mensagens de saída para plataformas de agentes externos. Um contêiner principal também é um servidor IIOP, ouvindo o endereço ACC (*Agent Communication Channel*) oficial da plataforma do agente para mensagens recebidas de outras plataformas. Os dois agentes de sistema obrigatórios, ou seja, o AMS (*Agent Management System*) e o DF (*Directory Facilitator*) padrão, são executados no contêiner principal (BELLIFEMINE *et al.*, 2001).

Novos contêineres de agente podem ser adicionados a uma plataforma JADE em execução; assim que um contêiner não principal é iniciado, ele segue um protocolo de registro simples com o principal e adiciona-se a uma tabela de contêineres mantida pelo principal. Os usuários podem empregar opções de linha de comando simples para saber em qual *host* e porta o *container* principal está escutando para novos registros de contêiner (BELLIFEMINE *et al.*, 2001).

O Java RMI é usada como o *middleware* de comunicação que faz o JADE assumir a aparência de uma plataforma única, mesmo que seja distribuída. Cada contêiner de agente exporta uma interface remota RMI para fornecer uma infraestrutura de objetos distribuídos para operações de plataforma, como gerenciamento de ciclo de vida do agente (BELLIFEMINE *et al.*, 2001).

Por exemplo, quando o AMS é solicitado a suspender um agente, pode muitas vezes ser o caso de que o agente está sendo executado em um contêiner diferente e isso é desconhecido para todos os agentes, AMS incluído, porque o endereçamento no nível do agente vê um JADE plataforma como um todo, seguindo as especificações

de nomenclatura do FIPA. O AMS só pode confiar na infraestrutura JADE subjacente para encontrar o agente relevante, portanto, ele simplesmente chama um método *suspendAgent*, passando o nome do agente como um argumento. Este método produzirá uma chamada local se o agente a suspender residir no *container* principal ou, se não for o caso, uma chamada remota RMI (BELLIFEMINE *et al.*, 2001).

2.4 Sistemas web

Aplicações web são sistemas desenvolvidos utilizando linguagens de programação como JavaScript e HTML e hospedados na web para serem executados no navegador (AL-FEDAGHI, 2011). Essas aplicações são acessadas por meio de uma conexão de rede e não precisa estar instalado na memória do dispositivo (TECHOPEDIA, 2021).

Os sites tradicionais introduzidos no começo da Internet forneciam páginas estáticas ao cliente, utilizando HTML e CSS, ou seja, sites com número fixo de páginas e sem interações com o usuário. Porém, com o passar dos anos e com a evolução das tecnologias, as pessoas necessitaram de sites dinâmicos com iterações e acesso a uma camada de persistência de dados, e com isso, tecnologias como Javascript começaram a ser utilizados (JADHAV *et al.*, 2015).

Para atingir a mesma experiência que os usuários possuem em aplicações *desktop*, surgiu a *Single Page Application* (SPA). Uma SPA é composta por componentes individuais que podem ser atualizados e substituídos, sem a necessidade de recarregar a página a cada ação do usuário, capacitando sistemas web a serem leves e robustos (JADHAV *et al.*, 2015).

Existem diversas bibliotecas e *frameworks* Javascript para o desenvolvimento web de uma SPA, como Angular [2], React [3] e Vue [4]. O React é uma biblioteca Javascript mantida pelo Facebook com foco em criar interfaces de usuário interativas em páginas web, renderizando apenas os componentes em que tiveram alterações de dados (REACT, 2021).

Para o desenvolvimento de uma aplicação em React, existem diversas IDEs, mas a principal e amplamente utilizada atualmente é o Visual Studio Code, desenvolvido

[2] https://angular.io.
[3] https://reactjs.org.
[4] https://vuejs.org.

pela Microsoft. Essa IDE é um editor de código-fonte leve, disponível para Windows, MacOS e Linux, e possui suporte interno para Javascript e Typescript (CODE, 2021).

2.5 Computação em nuvem

A computação em nuvem é uma plataforma de serviços que fornece acesso rápido a recursos computacionais por meio da internet, definindo o preço desses recursos conforme o uso (AWS, 2021e). É um paradigma que permite acesso ubíquo a um conjunto de recursos computacionais como servidores, banco de dados e funções e é composto por cinco características essenciais, três modelos de serviço e quatro modelos de implantação (MELL *et al.*, 2011). A seguir, são listadas as características e os modelos:

1. Características essenciais:
   - **Autoatendimento sob demanda:** Um consumidor pode fornecer recursos de computação, como tempo de servidor e armazenamento de rede sob demanda, sem a necessidade de interação humana com cada provedor de serviço;
   - **Amplo acesso à rede:** Os recursos estão disponíveis na rede e são acessados por meio de mecanismos que promovem o uso por meio de plataformas como computadores, celulares e *tablets*;
   - **Agrupamento de recursos:** Os recursos de computação do provedor são agrupados para atender a vários consumidores usando um modelo multilocatário, com diferentes recursos físicos e virtuais atribuídos e reatribuídos dinamicamente de acordo com a demanda do consumidor. Há uma sensação de independência de localização em que o cliente geralmente não tem controle ou conhecimento sobre a localização exata dos recursos fornecidos, mas pode ser capaz de especificar a localização em um nível mais alto de abstração (por exemplo, país, estado ou datacenter). Exemplos de recursos incluem armazenamento, processamento, memória e largura de banda da rede;
   - **Elasticidade rápida:** As capacidades podem ser elasticamente provisionadas e liberadas, em alguns casos automaticamente, para escalar rapidamente para fora e para dentro de acordo com a demanda. Para o

consumidor, os recursos disponíveis para provisionamento muitas vezes parecem ser ilimitados e podem ser apropriados em qualquer quantidade a qualquer momento;

- **Serviço medido:** Os sistemas em nuvem controlam e otimizam automaticamente o uso de recursos, alavancando uma capacidade de medição1 em algum nível de abstração apropriado para o tipo de serviço (por exemplo, armazenamento, processamento, largura de banda e contas de usuário ativas). O uso de recursos pode ser monitorado, controlado e relatado, proporcionando transparência tanto para o provedor quanto para o consumidor do serviço utilizado.

2. Modelos de serviço:
   - **Software as a Service (SaaS):** O recurso fornecido ao consumidor é usar os aplicativos do provedor em execução em uma infraestrutura em nuvem. Os aplicativos são acessíveis a partir de vários dispositivos clientes por meio de uma interface de cliente fino, como um navegador da web (por exemplo, e-mail baseado na web) ou uma interface de programa. O consumidor não gerencia ou controla a infraestrutura de nuvem subjacente, incluindo rede, servidores, sistemas operacionais, armazenamento ou mesmo recursos de aplicativos individuais, com a possível exceção de definições de configuração de aplicativos específicas do usuário;
   - **Platform as a Service (PaaS):** O recurso fornecido ao consumidor é implantar na infraestrutura de nuvem aplicativos criados ou adquiridos pelo consumidor, criados usando linguagens de programação, bibliotecas, serviços e ferramentas com suporte do provedor. O consumidor não gerencia ou controla a infraestrutura de nuvem subjacente, incluindo rede, servidores, sistemas operacionais ou armazenamento, mas tem controle sobre os aplicativos implantados e, possivelmente, definições de configuração para o ambiente de hospedagem de aplicativos.
   - **Infrastructure as a Service (IaaS):** A capacidade fornecida ao consumidor é fornecer processamento, armazenamento, redes e outros recursos de computação fundamentais onde o consumidor é capaz de implantar e executar software arbitrário, que pode incluir sistemas operacionais e aplicativos. O consumidor não gerencia ou controla a infraestrutura de

nuvem subjacente, mas tem controle sobre os sistemas operacionais, armazenamento e aplicativos implantados; e, possivelmente, controle limitado de componentes de rede selecionados (por exemplo, firewalls de host).

3. Modelos de implantação:
   - **Nuvem privada:** A infraestrutura de nuvem é fornecida para uso exclusivo por uma única organização composta por vários consumidores (por exemplo, unidades de negócios). Ele pode pertencer, ser gerenciado e operado pela organização, por terceiros ou por uma combinação deles, e pode existir dentro ou fora das instalações;
   - **Nuvem da comunidade:** A infraestrutura de nuvem é fornecida para uso exclusivo por uma comunidade específica de consumidores de organizações que compartilham preocupações (por exemplo, missão, requisitos de segurança, política e considerações de conformidade). Ele pode ser de propriedade, administrado e operado por uma ou mais organizações na comunidade, um terceiro, ou alguma combinação deles, e pode existir dentro ou fora das instalações;
   - **Nuvem pública:** A infraestrutura em nuvem é fornecida para uso aberto pelo público em geral. Ele pode ser de propriedade, administrado e operado por uma organização empresarial, acadêmica ou governamental, ou alguma combinação deles. Ele existe nas instalações do provedor de nuvem;
   - **Nuvem híbrida:** A infraestrutura de nuvem é uma composição de duas ou mais infraestruturas de nuvem distintas (privada, comunidade ou pública) que permanecem entidades únicas, mas são unidas por tecnologia padronizada ou proprietária que permite a portabilidade de dados e aplicativos (por exemplo, explosão de nuvem para balanceamento de carga entre nuvens).

Com a computação em nuvem, empresas podem ativar instantemente milhares de servidores e entregar resultados rapidamente, não necessitando de um planejamento com infraestruturas de TI, oportunizando a substituição de despesas com infraestrutura por baixos custos variáveis (AWS, 2019). Plataforma de serviços em nuvem, como

a *Amazon Web Services* (AWS) [5], *Google Cloud* [6] e a *Microsoft Azure* [7] possuem e mantém essa infraestrutura, enquanto o cliente provisiona e usa o que precisa por meio de uma aplicação web.

A AWS é uma plataforma em nuvem que oferece mais de 175 serviços em todo mundo (AWS, 2019). A AWS abrange 69 zonas de disponibilidade em 22 regiões geográficas em todo mundo, com planos anunciados para mais 16 zonas de disponibilidade.

Em 2006, a AWS iniciou a oferecer serviços de computação em nuvem. Atualmente, essa plataforma fornece uma infraestrutura altamente confiável, escalável e de baixo custo, que abastece centenas de milhares de empresas em 190 países (AWS, 2019). Além disso, a AWS possui um nível gratuito que inclui um milhão de chamadas de API durante até 12 meses (AWS, 2021c).

A AWS oferece uma quantidade maior de serviços em comparação com outros provedores de nuvem, como armazenamento de dados e tecnologias emergentes como *Machine Learning* e Inteligência Artificial (AWS, 2019). Para o presente trabalho, foram utilizados os serviços: DynamoDB, Lambda e AppSync.

2.5.1 Lambda

Lambda é um serviço que permite que um código seja executado sem gerenciar ou provisionar servidores. Esse serviço processa cada acionamento individualmente e executa o código paralelamente, escalando de acordo com a necessidade. Dessa forma, é necessário apenas escrever o código *back-end* e o serviço realiza toda a administração. Esse serviço também conta com um nível gratuito que inclui 1 milhão de solicitações gratuitas por mês e 400.000 GB/segundos de tempo de computação por mês (AWS, 2021d).

No entanto, para desenvolver o *back-end*, é necessário utilizar uma ferramenta que seja compatível com a Lambda. Para isso, foi utilizado o NodeJS, uma plataforma Javascript para criação de código *back-end* (MADSEN *et al.*, 2015) altamente performática e confiável (BUTTIGIEG *et al.*, 2015).

---

[5] https://aws.amazon.com.
[6] http://cloud.google.com.
[7] https://azure.microsoft.com.

2.5.2 DynamoDB

O DynamoDB é um serviço de banco de dados chave-valor com desempenho rápido e escalável. Ele replica automaticamente os dados em vários servidores, fornecendo alta disponibilidade e durabilidade. Além disso, pode suportar 10 trilhões de requisições por dia e 20 milhões de requisições por segundo (AWS, 2021b). O DynamoDB conta com um nível gratuito de uso, disponibilizando 25 GigaByte (GB) de armazenamento físico de dados e 1 GB de transferência de dados agregado com os serviços da AWS. Portanto, ele foi utilizado para a persistência dos dados do trabalho.

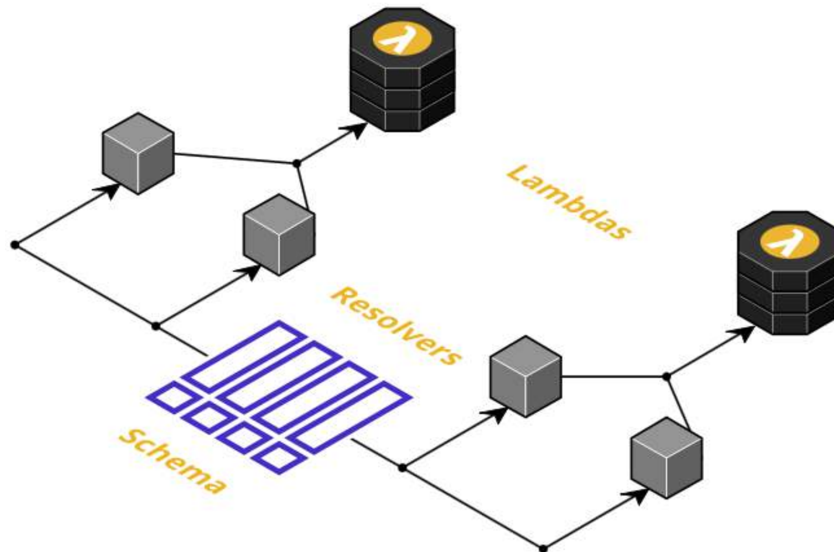
2.5.3 AppSync

Para que seja possível que o *front-end* e *back-end* possam se comunicar, é necessária uma API. Para isso, foi utilizado o AppSync, O AWS AppSync é um serviço totalmente gerenciado que facilita o desenvolvimento de APIs GraphQL a lidar com o trabalho pesado de se conectar com segurança a fontes de dados como AWS DynamoDB e Lambda (AWS, 2021a).

GraphQL é uma linguagem de consulta para APIs que fornece uma descrição completa e compreensível dos dados em sua API. Assim, torna-se mais fácil evoluir APIs ao longo do tempo e permite poderosas ferramentas para o desenvolvedor (GRAPHQL, 2021).

A Figura 10 representa o Diagrama Geral de uma API GraphQL, mostrando como é o processo de uma requisição até a chamada de uma função lambda.

Figura 10 – Diagrama da API



Fonte: Autoria própria (2021)

No *schema* são definidos três tipos chamados “especiais": *Query*, *Mutation* e *Subscription*; no tipo *Query* são definidas todas as operações de busca de dados; no tipo *Mutation* são definidas as operações que fazem inserção, atualização ou remoção de dados; e no tipo *Subscription* são maneiras de criar e manter uma conexão em tempo real com o servidor.

Cada um desses tipos possuem métodos, portanto, no tipo *Query* há métodos que realizam listagem, no tipo *Mutation* há métodos para inserção/deleção/atualização e no tipo *Subscription* há assinaturas para os métodos do tipo *Query* e *Mutation*.

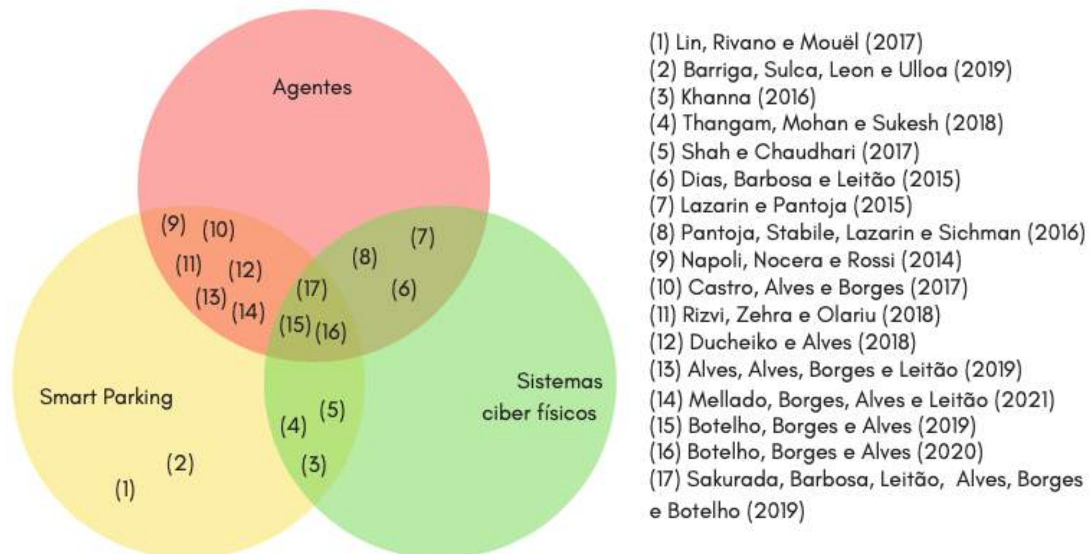
Cada método possui um *resolver* diferente e cada *resolver* está atrelado a uma função lambda. Um *resolver* irá organizar os parâmetros e executar a invocação da lambda. Assim, dependendo de qual método foi requisitado para a API é possível executar diferentes funções lambdas que realizam diferentes operações.

Com esse serviço, é possível criar uma API que permite a comunicação em tempo real, utilizando assinaturas (AWS, 2021a). Assim, este serviço foi utilizado para o desenvolvimento do trabalho pela necessidade de fazer requisições e atualizar as vagas do estacionamento em tempo real.

3 TRABALHOS RELACIONADOS

Existem vários trabalhos sendo desenvolvidos nas área de agentes embarcados e sistemas ciber-físicos, assim como trabalhos que utilizam sistemas ciber-físicos em Estacionamentos Inteligentes. Esta seção apresenta alguns destes trabalhos relacionados e suas principais características e contribuições. A Figura 11 mostra um diagrama de Venn que relaciona as principais áreas da presente pesquisa com seus respectivos trabalhos relacionados apresentados neste capítulo. No centro do diagrama de Venn é possível notar os trabalhos já publicados e que foram desenvolvidos no mesmo contexto do presente trabalho.

Figura 11 – Diagrama de Venn dos trabalhos relacionados



Fonte: Autoria própria (2021)

Em (LIN *et al.*, 2017) é realizado uma pesquisa no período de 2000 até 2016 em soluções para estacionamentos inteligentes. Os autores dividiram esta pesquisa em 3 temas: coleta de informação, desenvolvimento do sistema e disseminação de serviço. Para cada tema, é explicado as principais abordagem utilizadas nos trabalhos e resumidos seus objetivos e visões para resolver as dificuldades atuais encontradas no problema dos estacionamentos. Neste trabalho, os autores apresentam um guia compreensível para a pesquisa propondo soluções de estacionamentos inteligentes.

No trabalho desenvolvido por (BARRIGA *et al.*, 2019), apresenta uma revisão da literatura sobre tecnologias como componentes, sensores e softwares, empregadas em Estacionamentos Inteligentes. Os autores identificam os tipos mais utilizados de

cada componente e destacam as tendências de uso, além de analisar vários trabalhos cujo foco está no desenvolvimento de softwares para Estacionamentos Inteligente. Este artigo é utilizado como referência quanto às tecnologias e componentes que podem ser utilizados neste trabalho.

Em (KHANNA *et al.*, 2016) é proposto um estacionamento inteligente baseado em Internet das Coisas que utiliza um Raspberry Pi como um intermediário entre os sensores do estacionamento e um servidor de Cloud gerenciando as informações das reservas de vagas. O autor utiliza-se de um aplicativo móvel para auxiliar o motorista a saber quais vagas do estacionamento estão disponíveis. Com o sistema proposto, o autor apresenta uma alternativa para o problema dos estacionamentos com uma arquitetura física, integrada em nuvem.

Em (THANGAM *et al.*, 2018) é proposto um sistema de reservas para um estacionamento inteligente utilizando um Raspberry Pi para o reconhecimento óptico de caracteres para identificação da placa do carro e reconhecimento fácil do motorista para fornecer segurança. Para requisitar uma vaga, o motorista deve informar o local, o número da vaga, a placa do carro e uma foto para verificação facial. Assim que chega no estacionamento, a reserva é identificada através da verificação dos dados informados previamente.

Em (SHAH *et al.*, 2017) um sistema de estacionamento baseado em sensores ultrassônicos e Arduíno é proposto. Neste trabalho, os sensores ultrassônicos são conectados ao Arduíno e através do pulso de onda ultrassônica produzido pelo sensor, é possível detectar se há um carro estacionado ou não. Neste sistema, quando o sensor detecta uma vaga livre, é feita uma atualização em um sistema web, onde é possível visualizar todas as vagas do estacionamento.

Em (DIAS *et al.*, 2015) é descrito a implantação de agentes embarcados em um sistema de produção em pequena escala composto por robôs e controladores lógicos programáveis. Em uma das abordagens utilizadas, foi-se desenvolvidos agentes no *framework* JADE e embarcados em um Raspberry Pi para controle do processo físico do sistema. Os autores consideram esta abordagem completamente transparente, pois o desenvolvimento dos agentes não segue nenhum requisito especial e os mesmos funcionam perfeitamente no Raspberry Pi. Uma grande vantagem dessa abordagem é a possibilidade de utilização dos agentes para executar programas de controle em tempo real.

Em (LAZARIN *et al.*, 2015) é apresentado uma plataforma para embarcar agentes em plataformas de hardware. Esta plataforma consiste em embarcar o *framework* de agentes Jason em um Raspberry Pi para controlar os dispositivos de hardware e um Arduino para controlar os sensores e atuadores. Para que haja uma comunicação entre o software e o hardware é necessário estabelecer uma comunicação entre eles, para isso, os autores desenvolveram a biblioteca Javino que é um protocolo de comunicação para troca de mensagens entre o Java e o Arduino através de uma porta serial. Como resultado , é apresentado um chassi de um veículo capaz de se mover e desviar de veículos utilizando o Javino.

Em (PANTOJA *et al.*, 2016) é proposta uma arquitetura para programação de agentes embarcados utilizando o *middleware* Javino e filtros de percepção denominada ARGO. Os autores propõem a aplicação de filtros de percepção para reduzir o gargalo no ciclo de percepções dos agentes quando a percepção dos mesmos aumenta, reduzindo o custo de processamento. Como resultado, os autores apresentam experimentos utilizando uma plataforma de veículo terrestre em um cenário de colisão. Utilizando o ARGO, foi possível concluir que o uso de filtros de percepção foi capaz de prevenir colisões de forma eficaz.

Em (DI NAPOLI *et al.*, 2014a) um sistema para alocação de vagas baseado em um mecanismo de negociação entre agentes de software é apresentado. Neste sistema, os motoristas negociam as vagas diretamente com um agente administrador que considera como base a distância e custo da vaga. Os autores também apresentam um protocolo de negociação específico para o sistema desenvolvido baseado em rodadas, propostas e contrapropostas.

Em (RIZVI *et al.*, 2018), é utilizado um agente em nuvem, chamado *parker*, que define uma vaga de estacionamento que seja mais apropriada para um motorista. No sistema, cada veículo é um agente que possui como crença as características de seu motorista. O agente *parker*, na nuvem, é capaz de interagir com serviços de tráfego, clima, e emergência para encontrar a melhor vaga disponível no estacionamento com base nas requisições feitas pelo agente motorista.

3.1 Trabalhos do Grupo de Pesquisa

Estes trabalhos fazem parte do projeto Smart Parking, que em parceria com o IPB, possui o objetivo de desenvolver um estacionamento inteligente com uma abordagem baseada em agentes devido a capacidade de inteligência distribuída dos agentes e pela escalabilidade oferecida.

(CASTRO *et al.*, 2017) desenvolveu um SMA utilizando o *framework* JaCaMo com o intuito de gerenciar vagas em um estacionamento inteligente. Este sistema é composto por dois tipos de agentes: *manager* que é responsável pela administração das vagas; e *drivers* que representam os motoristas querendo utilizar o estacionamento. As vagas do estacionamento são atribuídas aos motoristas de acordo com um grau de confiança que cada motorista possui. Os resultados mostraram, em diversas simulações, o funcionamento do sistema e que motoristas que possuem maior grau de confiança recebem vaga em menor tempo.

Em (DUCHEIKO *et al.*, 2018) é proposto um modelo de raciocínio e um protocolo de negociação para implementar um mecanismo de negociação descentralizado. O modelo adota apenas um tipo de agente, o motorista, que é capaz de assumir tanto o papel de vendedor, como de comprador de uma vaga. A principal característica desse trabalho, segundo os autores, é a não existência de um agente centralizador no sistema, assim, todos os agentes independem de um módulo central para obter uma vaga.

Em (ALVES *et al.*, 2019) é apresentado um estudo a respeito de protocolos de negociação para problemas de consenso em estacionamentos inteligentes. Algumas estratégias de negociação, como o protocolo *Contract Net*, o leilão inglês e o leilão holandês foram implementados utilizando o *framework* JADE para evidenciar a melhor estratégia para aplicar em um SMA aplicado à estacionamentos inteligentes.

Em (MELLADO *et al.*, 2021) propõe um método de precificação dinâmico para promover alocação de vagas em um estacionamento e a diminuição do congestionamento nas cidades. Para isso, um SMA foi implementado utilizando o *framework* JaCaMo, visando alterar dinamicamente os preços das vagas com base em características observadas no ambiente. Segundo as simulações realizadas, os autores concluíram que a utilização do método de precificação dinâmico aumenta a ocupação média do estacionamento mantendo o ganho total.

Em (BOTELHO, P. *et al.*, 2019), trabalho em fase inicial, é proposto uma ar-

quitetura utilizando agentes embarcados em um Raspberry Pi e sensores conectados a ESP-12e como uma alternativa para os estacionamentos inteligentes. Os autores propõem utilizar um agente JADE como gerenciador das vagas, um SMA responsável pela parte de negociação com o motorista e o ESP-12e para capturar o estado da vaga e enviar para o Raspberry via protocolo MQTT. As principais vantagens desta arquitetura, segundo os autores, está no baixo custo de instalação, visto que as placas de maior custo são utilizadas apenas sob demanda do estacionamento.

Em (SAKURADA *et al.*, 2019) uma arquitetura ciber-física para estacionamentos inteligentes é apresentada, porém, esta arquitetura considera um estacionamento para bicicletas. Cada vaga possui um Raspberry Pi com um agente embarcado, o qual controla um dispositivo físico capaz de liberar o acesso a vaga da bicicleta.

Em (BOTELHO, P. W. *et al.*, 2020) descrevem a implementação de um aplicativo móvel e uma API na nuvem como uma solução para integrar os componentes presentes em um estacionamento inteligente. Com base nos experimentos realizados, a arquitetura propõe versatilidade ao permitir que qualquer dispositivo móvel realize uma requisição, necessitando apenas de do aplicativo e acesso a internet. Os experimentos ainda mostram que esta arquitetura é flexível pois, graças a computação em nuvem, é possível aplicá-la utilizando diferentes componentes na arquitetura.

3.2 Considerações finais

Neste capítulo foram apresentados os principais conceitos acerca de Cidades e Estacionamentos Inteligentes, Sistemas Ciber-físicos, Agentes, Sistemas Multi-Agentes e Computação em Nuvem. Por fim, foram apresentados os principais trabalhos relacionados acerca de Agentes Inteligentes, Sistemas Ciber-Físicos e Computação em Nuvem. Ainda foram apresentados os trabalhos desenvolvidos pelo grupo de pesquisa e os artigos publicados no mesmo contexto do presente trabalho.

Como é possível observar os trabalhos supracitados utilizam de maneira isolada diferentes componentes que fazem parte de um estacionamento inteligente. A partir disso, constata-se a necessidade de implantar uma integração entre todos os componentes para a devida construção de um protótipo ciber-físico para estacionamentos inteligentes.

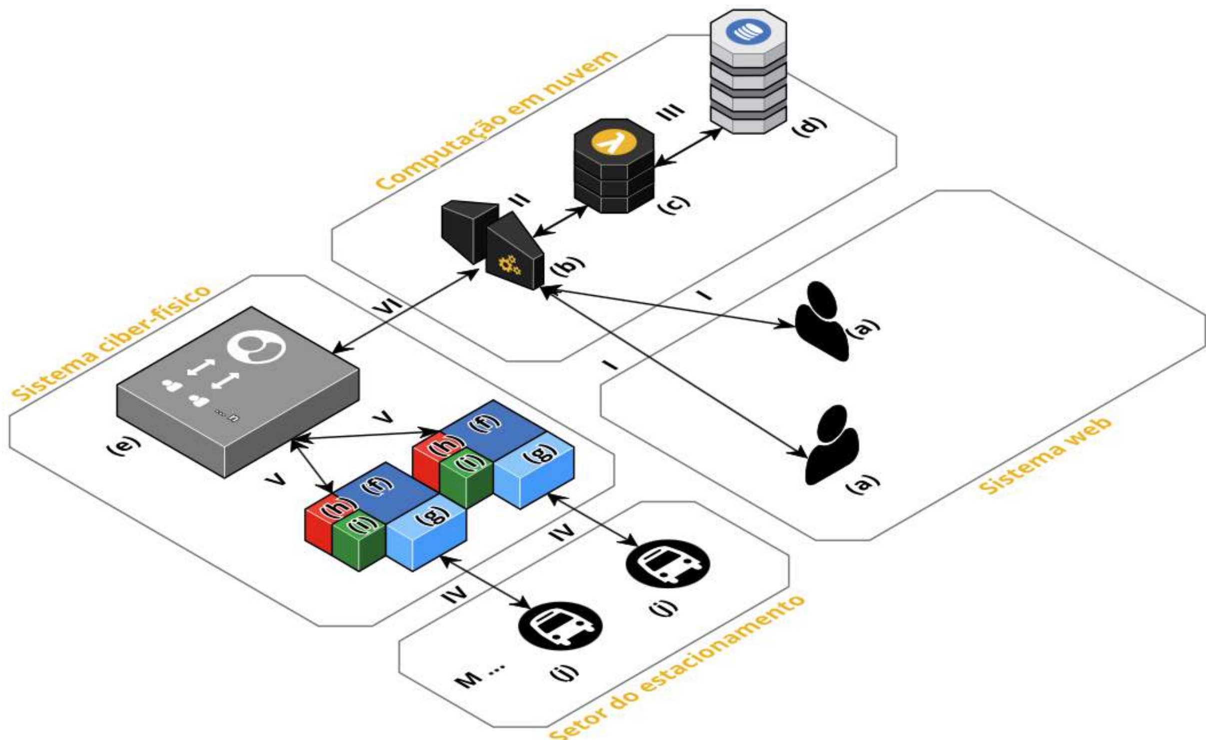
4 DESENVOLVIMENTO DO SISTEMA

Neste capítulo é apresentado o desenvolvimento de todos os componentes do sistema. A Seção 4.1 descreve uma visão geral da arquitetura e como cada um dos componentes se comunica. A seção 4.2 apresenta o fluxo do gerenciador de estacionamento e como são manipulados as comunicações com o sistema em nuvem. A seção 4.3 descreve como foi implementada a arquitetura em nuvem que armazena os dados das vagas e realiza a comunicação entre o gerenciador de estacionamento e o sistema ciber-físico. A seção 4.4 descreve a implementação dos agentes no sistema ciber-físico e a comunicação do hardware com a nuvem.

4.1 Arquitetura

A arquitetura de um sistema se refere a um conjunto de estruturas necessárias para dar logicidade ao sistema, especificando as propriedades e relacionamentos dos componentes envolvidos (CLEMENTS *et al.*, 2002). Portanto, esta seção apresenta o diagrama geral da arquitetura com todos os serviços utilizados e suas comunicações.

Figura 12 – Visão geral da arquitetura



Fonte: Autoria própria (2021)

Uma visão geral da arquitetura e dos componentes utilizados é apresentada na

Figura 12. A arquitetura contém: um módulo de Computação em Nuvem utilizado para replicar mensagens trocadas entre os componentes e armazenar as informações das vagas do estacionamento; um sistema para gerenciamento do estacionamento capaz de realizar requisições de vagas, receber mensagens do sistema ciber-físico, e ilustrar o estado real do estacionamento com as vagas disponíveis e indisponíveis; um sistema ciber-físico responsável por detectar a presença de um veículo na vaga e atualizar a vaga em tempo real utilizando um SMA. Abaixo estão enumerados cada componente da Figura 12.

(a) Os usuários administrador ou motorista;
(b) A API GraphQL no Appsync;
(c) As funções lambdas escritas em NodeJS;
(d) O banco de dados no DynamoDB;
(e) Um Raspberry Pi com um SMA em JADE;
(f) Um módulo ESP-12e;
(g) Sensores ultrassônicos modelo hc-sr04;
(h) Um led vermelho;
(i) Um led verde;
(j) As vagas do estacionamento;

Na operação I, o usuário (a) pode requisitar todas as vagas disponíveis no estacionamento, assim como requisitar uma vaga para estacionar. A API então recebe e processa essa requisição e na operação II, a API (b) dispara uma função lambda (c) para ser executada. Na operação III, a lambda se comunica com o banco de dados (d) para realizar uma operação busca/inserção/atualização/deleção, dependendo do que foi requisitado.

Na operação IV, o sensor ultrassônico (g) que está conectado a um ESP-12e (f) detecta a presença/não presença de um carro na vaga (j), havendo assim uma atualização na vaga (j). Na operação V, o ESP-12e (f) envia uma mensagem para o Raspberry Pi (e) sobre essa atualização. Na operação VI, o Raspberry Pi (e) se comunica com a API (b) informando o novo estado da vaga. Por fim, novamente na operação V, o Raspberry Pi (e) informa ao ESP-12e (f) qual led deve acender, vermelho (h) ou verde (i) com base na nova informação da vaga. Ao mesmo tempo, por meio da operação I, a API (b) replica essa atualização de vaga para os usuários (a).

A partir da visão geral da arquitetura é possível especificar cada componente

com suas funcionalidade e papéis dentro do sistema. Nas próximas seções serão descritas as especificações de cada componente.

4.2 Sistema de gerenciamento do estacionamento

O sistema de gerenciamento do estacionamento foi desenvolvido utilizando a biblioteca React. De maneira geral, o sistema possui um controle de usuário, podendo ter dois tipos, o administrador e o motorista que realizam diferentes operações dentro do sistema. A seção 4.2.1 detalha os usuários e suas permissões e a seção 4.2.2 detalha como o sistema realiza as operações.

4.2.1 Usuários

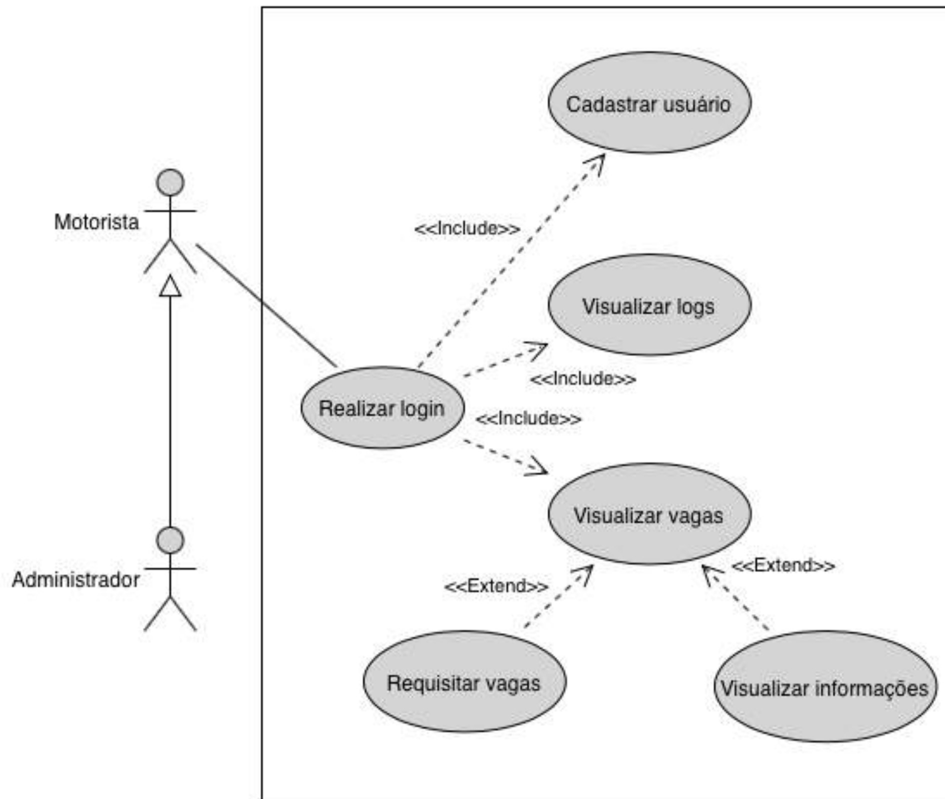
Dois grupos de usuários foram definidos para acesso ao sistema: administrador e motorista. O motorista pode realizar seu cadastro na tela inicial e possui a função de visualizar as vagas disponíveis e requisitar uma vaga para estacionar. Por ser um protótipo, o desenvolvimento de funcionalidades como cadastro e manutenção de usuários não foram implementadas.

O administrador possui acesso à criação de novos usuários administradores, um visualizador geral do estacionamento com informações de quem está atualmente na vaga, qual o horário de entrada e saída e também possui acesso aos *logs* do sistema, ou seja, o usuário administrador consegue visualizar todas as requisições realizadas por motoristas e as respostas dadas por meio do SMA e do sistema ciber-físico.

Para representar as permissões de acesso dos usuários no sistema, foi criado um diagrama de caso de uso. Diagramas de Caso de Uso são modelos que faz parte da linguagem de documentação e modelagem denominada *Unified Modeling Language*. A utilização destes diagramas permitem uma visão geral e compreensão do sistema a ser desenvolvido.

Assim, o diagrama de caso de uso do sistema é apresentado na Figura 13.

Figura 13 – Diagrama de Caso de Uso



Fonte: Autoria própria (2021)

No presente trabalho, os atores são identificados como motoristas e administradores. Os atores administrador e motorista possuem relacionamento de generalização, ou seja, o administrador herda todos os atributos do motorista, possibilitando interagir com os casos de usos em que o motorista está se relacionando.

Os casos de uso presentes no diagrama são:

1. Realizar login: tanto o motorista quanto o administrador precisam estar logados para ter acesso as funcionalidades do sistema;
2. Cadastrar usuários: tanto o motorista quanto o administrador podem realizar cadastros de usuários;
3. Visualizar vagas: funcionalidade que permite visualizar as vagas do estacionamento;
4. Requisitar vaga: funcionalidade onde o motorista pode requisitar uma vaga ao sistema;
5. Visualizar informação: funcionalidade onde o administrador pode visualizar os dados do motorista vinculado a vaga assim como o estado atual da vaga;
6. Visualizar *logs*: funcionalidade onde o administrador pode visualizar os todos

os *logs* de requisições do sistema;

4.2.2 Funcionamento geral

A tela principal do sistema, para ambos os usuários, é a tela que contém as vagas. Portanto, a primeira requisição feita após o login do usuário é a requisição das vagas. A Figura 14 mostra a requisição sendo feita ao iniciar a aplicação.

Figura 14 – Requisição das vagas ao iniciar o sistema

```
1  useEffect(() => {
2      void getAllSpots((error, result) => {
3          if (result && result.getAllSpots) {
4              setAllSpots(result.getAllSpots.sort(sortAlphaNumber));
5          } else if (error) {
6              console.error(error);
7          }
8      });
9  }, [setAllSpots]);
```

Fonte: Autoria própria (2021)

Na linha 2 a função *getAllSpots* (disponível na Figura 15) é invocada e uma função de *callback* é passado como parâmetro para o método. Assim, quando a requisição da função terminar, a função de *callback* será executada e então, se houver um resultado válido com todas as vagas, a linha 4 irá salvar este resultado em uma variável, ordenando as vagas por ordem alfabética, para que seja exibido na tela todas as vagas. Caso não houver resultados válidos, na linha 6 é apresentado um erro.

Figura 15 – Função que faz a requisição das vagas para a API

```
1  import graphql, { GraphQLCallback, GraphQLReturn } from 'utils/graphql';
2  import * as Types from '../api/types';
3  import { getAllSpots as getAllSpotsQuery } from '../graphql/queries';
4
5  type GetAllSpotsType = {
6      getAllSpots: Types.Spot[];
7  };
8
9  export default async function getAllSpots(
10     callback: GraphQLCallback<GetAllSpotsType>
11 ): GraphQLReturn<GetAllSpotsType> {
12     return graphql<GetAllSpotsType>(
13         {
14             query: getAllSpotsQuery,
15         },
16         callback
17     );
18 }
19
```

Fonte: Autoria própria (2021)

Todas as funções que realizam uma requisição para a API invocam uma função genérica que serve apenas para lidar com as requisições, importada na linha 1 da Figura 15. Essa função e todos os outros códigos estão disponíveis no github [1]. Portanto, a função *getAllSpots* executa essa função genérica passando como parâmetro a *Query* e o *callback* recebido como parâmetro.

A Figura 16 mostra a função que é chamada quando o usuário motorista deseja requisitar uma vaga.

[1] https://github.com/laca-is/embedded-parking.

Figura 16 – Função para requisitar uma vaga no estacionamento

```
1  function onRequestSpot({ status, veihicle, ...spot }: Types.Spot) {
2      showMessage('Requisitando vaga...', 'info');
3
4      const today = new Date();
5
6      const input: Types.RequestSpotInput = {
7          ...spot,
8          driverId,
9          startTime: format(today, 'kk:mm'),
10         endTime: format(addHours(today, 1), 'kk:mm'),
11         date: formatISO(today, { representation: 'date' }),
12     };
13
14     void requestSpot(input, (error, data) => {
15         if (data) {
16             showMessage('Aguarde a negociação.', 'info');
17         } else if (error) {
18             showMessage('Ocorreu um erro na requisição.', 'error');
19         }
20     });
21 }
```

Fonte: Autoria própria (2021)

Na linha 6, são definidos os parâmetros que serão enviados na requisição. O *driverId* com a informação do motorista que está requisitando a vaga e o *date*, *startTime* e *endTime* com a informação da data e hora que o motorista estará ocupando a vaga.

Na linha 14, a função *requestSpot* é chamada, seguindo a mesma lógica da Figura 15, porém, importando a *mutation* de *requestSpot* gerada anteriormente. Nesse momento, se houver uma resposta, o sistema iniciará a negociação da vaga. Para esse sistema, a negociação da vaga está integrado e receberá a requisição da negociação por meio de uma *subscription*, como mostra a Figura 17.

Figura 17 – Assinatura da função de requisição de vagas

```
1  subscriptions.subscribe<Types.Spot>({ onRequestSpot }, (response) => {
2      if (response) {
3          const { status, veihicle, ...spot } = response;
4          updateSpot(response);
5          negotiate(spot as Types.RequestSpotInput);
6      }
7  });
```

Fonte: Autoria própria (2021)

Quando a requisição é recebida, por meio da *subscription*, o sistema irá chamar a função de negociação, disponível na Figura 18, onde é gerado um número aleatório de

0 a 1. Se o número gerado for menor que 0.85 a negociação será bem sucedida, caso contrário, o sistema irá rejeitar a requisição da vaga, obrigando o motorista e requisitar outra vaga. A negociação ocorre desta maneira pois este trabalho é um protótipo e por isso não foi integrado a um sistema que implemente protocolos de negociação, portanto, o motorista possui uma chance de 85% da sua negociação ser bem sucedida.

Figura 18 – Função da negociação da vaga

```
1  const negotiate = useCallback((spot: Types.RequestSpotInput) => {
2      setTimeout(() => {
3          const acceptNegotiation = Math.random() < 0.85;
4
5          if (acceptNegotiation) {
6              void requestSuccess(spot, (error, _) => {
7                  if (error) {
8                      console.error(error);
9                  }
10             });
11         } else {
12             void requestFailed(spot.spot, (error, _) => {
13                 if (error) {
14                     console.error(error);
15                 }
16             });
17         }
18     }, 5000);
19 }, []);
```

Fonte: Autoria própria (2021)

Agora, com a requisição da vaga feita o motorista já pode estacionar o carro na vaga que foi requisitada. Assim, quando o carro chegar na vaga, o sistema ciber-físico irá chamar o método *onCarArrived* (mais detalhes na Seção 4.4) que será recebido no sistema de gerenciamento por meio da *subscription* ilustrada na Figura 19.

Figura 19 – Função de assinatura quando um carro chega na vaga

```
1  subscriptions.subscribe<Types.Spot>({ onCarArrived }, (response) => {
2      if (response) {
3          updateSpot(response);
4      }
5  });
```

Fonte: Autoria própria (2021)

Após receber essa requisição vindo do sistema ciber-físico, a vaga em questão irá ser atualizada no sistema, por meio do método *updateSpot* disponível na Figura 20,

mantendo assim o sistema atualizado em tempo real.

Figura 20 – Função para atualizar a vaga recebida

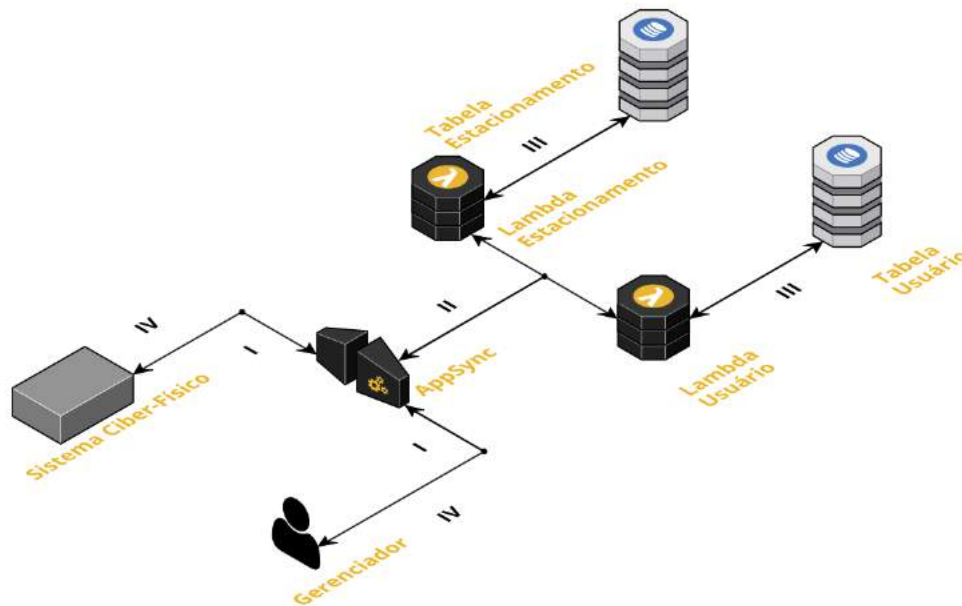
```
1  const updateSpot = useCallback(
2      (spotUpdated: Types.Spot) => {
3          setAllSpots((previous) =>
4              previous?.map((item) =>
5                  item.spot === spotUpdated.spot ? spotUpdated : item
6              )
7          );
8      },
9      [setAllSpots]
10 );
```

Fonte: Autoria própria (2021)

4.3 Computação em nuvem

A Figura 21 representa o Diagrama Geral do sistema e nuvem, mostrando como os componentes e serviços se relacionam e como os dados do estacionamento são coletados e armazenados. Esse diagrama possui operações numeradas que serão descritas a seguir.

Figura 21 – Diagrama geral do sistema em nuvem



Fonte: Autoria própria (2021)

A operação I ilustra uma requisição para a uma API GraphQL dada por meio do usuário motorista/administrador ou do sistema ciber-físico. O usuário pode realizar

essa requisição fazendo login ou requisitando uma vaga e o sistema ciber-físico para atualizar o estado atual da vaga em que ele representa.

Cada requisição feita via operação I resultará na operação II onde é executado uma função FaaS [2]. Nessa operação, a função lambda recebe os parâmetros da API para executar a requisição do usuário. Nessas funções estão concentradas toda a lógica de negócio do sistema que serão apresentadas na subseção 4.3.2.

Na operação III, a função realiza a comunicação com o banco de dados, podendo executar listagem, inserção, atualização e remoção de registros. No banco de dados estão armazenados as informações do estacionamento e dos motoristas. Após receber a resposta do banco de dados, a função manipula os dados recebidos do banco de dados. Na operação IV a função retorna uma resposta para quem fez a requisição e replica a resposta para todos que assinaram aquela requisição.

4.3.1 Banco de dados

Para este trabalho foram criadas duas tabelas no DynamoDB, uma para armazenar as informações dos usuários administradores e motoristas (*user*) e outra para armazenar as vagas do estacionamento e as informações de cada vaga (*parking*).

Todas as vagas do estacionamento estão pré mapeadas na tabela *parking*, portanto, conforme os motoristas vão requisitando as vagas, os dados de cada vaga são atualizados. Da mesma forma, quando um motorista entra/sai do estacionamento, os dados do motorista são vinculados/desvinculados a vaga. Todos os serviços do sistema consomem os dados dessa tabela, mantendo-se assim sempre atualizados.

Para cada item da tabela são armazenados os seguintes atributos:

1. *spot*: o setor e o número da vaga;
2. *date*: a data em que a vaga está ocupada;
3. *driverId*: a informação do motorista estacionado na vaga;
4. *startTime*: a hora que o motorista reservou a vaga;
5. *endTime*: a data que o motorista sairá da vaga;
6. *price*: o preço pago por aquela vaga;
7. *status*: o estado atual da vaga.

[2] *Function as a Service*, ou FaaS, é um serviço de computação em nuvem que permite executar código sem precisar gerenciar servidores (AWS, 2021d).

Cada vaga pode assumir os valores apresentados abaixo. As vagas possuem 4 *status* no sistema de gerenciamento do estacionamento, porém, na parte física é assumido apenas os valores de vaga livre e ocupada.

- *FREE:* Indica que a vaga está livre;
- *IN_NEGOTIATION:* Indica que a vaga está em negociação;
- *WAITING_FOR_PARKING:* Indica que a vaga foi negociada com sucesso e está aguardando o motorista estacionar;
- *BUSY:* Indica que o motorista já estacionou e a vaga está ocupada.

4.3.2 Funções lambdas

Para este trabalho foram desenvolvidas duas funções lambdas utilizando a linguagem Javascript, uma para controle de usuário (*user*) e outra para o controle do estacionamento (*parking*). Assim que um método é requisitado para a API, ela precisa identificar qual é a lambda vinculada àquele método. Para isso, é necessário mapear todos os métodos que foram definidos no *schema graphql* dentro do *resolver*. A Figura 22 mostra como é feito o mapeamento do método *getAllSpots*.

Figura 22 – Mapeamento do método *getAllSpots*

```
1  - type: Query
2    dataSource: parking
3    field: getAllSpots
4    request: requests/request.vtl
5    response: responses/response.vtl
```

Fonte: Autoria própria (2021)

Na linha 1, é definido que esse método é do tipo *Query*, na linha 2 é identificada a lambda em que esse método está atrelado, na linha 3 é definido o nome do método o qual deve ser igual ao que foi definido no *schema graphql* e nas linhas 4 e 5 são definidos os códigos que serão executados antes e depois da invocação da lambda.

Para que cada método dentro da lambda seja organizado, o código da Figura 23 é executado sempre antes de todas as requisições. Este código é apenas um meio de organizar os parâmetros passados para a requisição antes que a lambda seja executada.

Figura 23 – Código executado antes da lambda

```
1   #set( $event = {})
2   #set( $arguments = {})
3   $util.qr( $event.put("field", $context.info.fieldName) )
4   #foreach ($key in $context.args.keySet())
5       $util.qr( $arguments.put($key, $context.args.get($key)) )
6   #end
7   $util.qr( $event.put("arguments",  $arguments) )
8   {
9       "version" : "2018-05-29",
10      "operation": "Invoke",
11      "payload": $util.toJson( $event )
12  }
```

Fonte: Autoria própria (2021)

Nas linhas 1 e 2 são criados os objetos *event* e *arguments* respectivamente. No objeto *event*, estará definido qual é o nome do método por meio da propriedade *field* (linha 3) e os argumentos por meio do objeto *arguments*. Dentro do objeto *arguments* estarão todos os parâmetros que foram passados na invocação do método pela API (linhas 4 a 6). Dessa forma, na linha 11, o objeto *event* é passado como argumento para a lambda, e será um objeto de argumento mais organizado, como mostra a Figura 24.

Figura 24 – Exemplo de argumento para o método *getAllSpots*

```
1 {
2     "field": "getAllSpots",
3     "arguments": {}
4 }
5
```

Fonte: Autoria própria (2021)

A Figura 25 ilustra a primeira instrução que é executada pela lambda de *parking*. O *switch* executado escolhe qual método dentro da função lambda será executado com base no atributo *field* definido anteriormente.

Figura 25 – Operador *switch* na lambda *parking*

```
1  switch (event.field) {
2      case 'insertParkingSpots':
3          return insertParkingSpots(event.arguments);
4      case 'getSpotsByStatus':
5          return getSpotsByStatus(event.arguments);
6      case 'getAllSpots':
7          return getAllSpots();
8      case 'requestSpot':
9          return requestSpot(event.arguments);
10     case 'requestSuccess':
11         return requestSuccess(event.arguments);
12     case 'requestFailed':
13         return requestFailed(event.arguments);
14     case 'carArrived':
15         return carArrived(event.arguments);
16     case 'carLeft':
17         return carLeft(event.arguments);
18     case 'getHistory':
19         return getHistory(event.arguments);
20     default:
21         throw new Error('Method not allowed');
22 }
```

Fonte: Autoria própria (2021)

No caso do método *getAllSpots*, a partir do *switch*, a linha 7 será executada, chamando então uma nova função ilustrada na Figura 26.

Figura 26 – Função *getAllSpots*

```
1  async function getAllSpots() {
2      const result = await dynamo
3          .scan({
4              TableName: 'parking',
5          })
6          .promise();
7
8      if (!result.Items.length) {
9          return {
10             errorType: 'NO_RESULT_FOUND',
11             errorMessage: 'Nenhuma vaga foi encontrada.'
12         }
13     }
14
15     return result.Items.map((item) => DynamoDB.Converter.unmarshall(item));
16 }
```

Fonte: Autoria própria (2021)

A função *getAllSpots* irá buscar na tabela *parking* do DynamoDB todas as vagas do estacionamento e irá retornar, para quem fez a requisição, uma lista contendo as vagas (linha 15). Caso nenhuma vaga seja encontrada, um erro é retornado (linha 9).

Do mesmo modo, todos os outros métodos definidos no *schema graphql* irão seguir o mesmo processo descrito anteriormente. A Figura 27 mostra o objeto *event* gerado para o método *requestSpot* com os argumentos passados para a requisição.

Figura 27 – Exemplo de argumento para o método *requestSpot*

```
1  {
2      "field": "requestSpot",
3      "arguments": {
4          "input": {
5              "spot": "A1",
6              "driverId": "063.820.049-01",
7              "startTime": "14:00",
8              "endTime": "18:00",
9              "date": "2021-10-28"
10         }
11     }
12 }
13
```

Fonte: Autoria própria (2021)

Assim, após o *switch* da Figura 25, a função *requestSpot* será chamada e a vaga requisitada será atualizada para o status de em negociação, como mostra a Figura 28.

Figura 28 – Função *requestSpot*

```
1  async function requestSpot(args) {
2      const payload = {
3          ...args.input,
4          status: 'IN_NEGOTIATION',
5      };
6
7      return dynamodb
8          .put({
9              TableName: 'parking',
10             Item: {
11                 ...payload,
12             },
13         })
14         .promise();
15 }
```

Fonte: Autoria própria (2021)

4.4 Sistema Ciber-Físico

O Sistema Ciber-Físico, conforme previamente ilustrado na Figura 12, é composto por dois elementos principais, um Raspberry Pi e um módulo ESP-12e. O Raspberry Pi é responsável por controlar um setor de um estacionamento por meio de um SMA em JADE, o módulo ESP-12e é responsável por detectar a presença de um carro na vaga de estacionamento e indicar visualmente por meio de *leds* o status atual da vaga.

Assim, quando um carro estaciona em uma vaga, o módulo ESP-12e detecta a presença do carro e envia uma mensagem para o Raspberry por meio de um *broker* que implementa o protocolo MQTT. Um agente gerente, no Raspberry, recebe essa mensagem e informa outro agente, o qual é responsável pela vaga, que possui um carro estacionado. Este agente vaga então replica essa informação para o banco de dados, informando o novo status da vaga. Assim, o mesmo agente repassa que o status da vaga foi atualizado no banco para o módulo ESP-12e, via MQTT, o qual acende o *led* correspondente ao status.

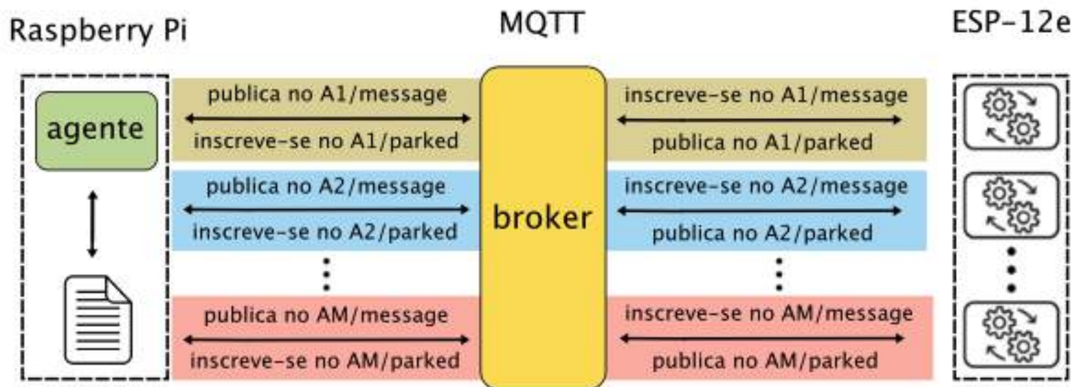
A subseção 4.4.1 apresenta o fluxo de configuração do *broker* MQTT e as mensagens trocadas entre o Raspberry e o ESP-12e. A subseção 4.4.2 entrará em detalhes sobre o fluxo de comunicação dos agentes no Raspberry e a subseção 4.4.3 irá abordar o fluxo de comunicação entre os dispositivos físicos presentes no módulo ESP-12e.

4.4.1 Protocolo MQTT

A comunicação entre o Raspberry e o módulo ESP-12e dá-se-a através de um *broker* MQTT. Este *broker* segue o princípio *Publish-Subscribe*, que tem seu funcionamento por meio de tópicos. De maneira geral, um dispositivo publica em um tópico no *broker* e o mesmo replica esta mensagem a todos os dispositivos que estão inscritos neste tópico. Assim, é possível que dois dispositivos troquem informações, o que atende as requisições necessárias para o funcionamento do sistema.

A Figura 29 ilustra os tópicos seguidos para a troca de mensagens e ainda em qual tópico cada dispositivo está inscrito e publica mensagens.

Figura 29 – Tópicos do broker



Fonte: Autoria própria (2021)

Quando o sistema é iniciado, o Raspberry do setor se inscreve nos tópicos de cada vaga e cada ESP-12e se inscreve no tópico relacionado a própria vaga. Desse modo, quando o Raspberry publica uma informação em um tópico de uma vaga, a vaga recebe essa informação e pode também publicar uma informação para o Raspberry receber.

Para melhor organização das mensagens foi criado um tópico para cada vaga com dois subtópicos: *message* e *parked*. Ao publicar no subtópico *parked*, o ESP-12e passa a informação de que um carro entrou/saiu da vaga, a qual será recebida pelo Rapsberry que está inscrito no tópico. Do mesmo modo, ao publicar no subtópico *message*, o Raspberry passa a informação que a vaga foi atualizada com sucesso, a qual será recebida pelo ESP-12e inscrito no tópico. Os passos a seguir ilustram um exemplo prático de trocas de mensagens.

Quando o carro estaciona na vaga 1 do setor A:

- O ESP-12e publica no tópico ‘A1/parked’ que um carro estacionou;
- O Raspberry recebe essa mensagem e faz o que será detalhado na subseção a seguir;
- O Raspberry então publica no tópico ‘A1/message’;
- O ESP-12e recebe a mensagem e acende o LED vermelho.

Quando o carro sai da vaga 1 do setor A:

- O ESP-12e publica no tópico ‘A1/parked’ que um carro não está mais estacionado;
- O Raspberry recebe essa mensagem e faz o que será detalhado na subseção 4.4.2;

- O Raspberry então publica no tópico ‘A1/message’;
- O ESP-12e recebe a mensagem e acende o LED verde.

Um protocolo foi estabelecido para que o Raspberry e o ESP-12e possam trocar mensagens por meio do broker. As seguintes palavras chaves foram utilizadas:

1. “busy": mensagem enviada pelo ESP-12e para o Raspberry quando um carro estaciona na vaga;
2. “free": mensagem enviada pelo ESP-12e para o Raspberry quando um carro sai do vaga;
3. “green": mensagem enviada pelo Raspberry para o ESP-12e para que o LED verde acenda.
4. “red": mensagem enviada pelo Raspberry para o ESP-12e para que o LED vermelho acenda.

O *broker* utilizado neste trabalho foi o Mosquitto por ser leve e de fácil uso. Para iniciar o *broker* basta digitar *mosquitto* e o *broker* ficará disponível para acesso via endereço de IP na porta padrão 1883, como mostra a Figura 30.

Figura 30 – *Broker* mosquitto em execução

```
pi@raspberrypi:~ $ mosquitto
1635609906: mosquitto version 1.5.7 starting
1635609906: Using default config.
1635609906: Opening ipv4 listen socket on port 1883.
1635609906: Opening ipv6 listen socket on port 1883.
```

Fonte: Autoria própria (2021)

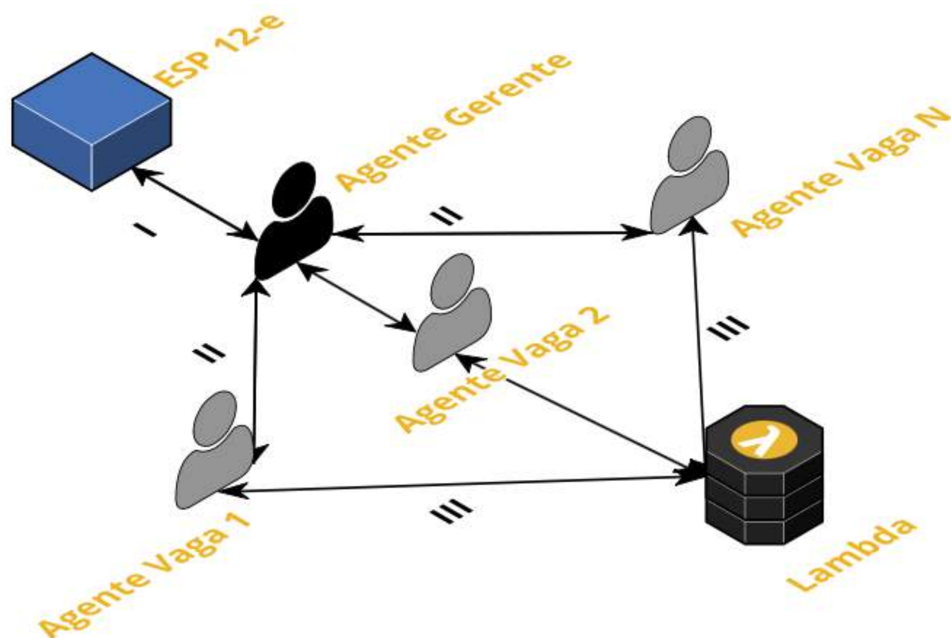
4.4.2 Raspberry Pi e Agentes

A Figura 31 apresenta o fluxo de comunicação dos agentes dentro do Raspberry Pi. Para isso, o diagrama demonstra o recebimento de uma mensagem através do ESP-12e até o momento em que o agente responsável pela vaga atualiza o *status* no banco de dados.

Cada Raspberry representa um setor de um estacionamento, o qual pode ter *N* setores com *N* Raspberries e cada setor pode ter *M* vagas, dependendo da demanda do estacionamento. Nesse caso, cada Raspberry então possui um agente gerente que será responsável por receber todas as mensagens via protocolo MQTT e M agente vagas, os quais receberão mensagens do agente gerente sobre o status da vaga e

então irão atualizar o status no banco de dados.

Figura 31 – Diagrama do Raspberry Pi



Fonte: Autoria própria (2021)

Na operação I, o ESP-12e envia uma mensagem para o agente gerente via protocolo MQTT informando que um carro acabou de estacionar/sair da vaga. Para isso, o ESP-12e publica no tópico MQTT correspondente a vaga a ser atualizada. O agente gerente então recebe essa mensagem, identifica qual é o agente responsável pela vaga e envia uma mensagem do tipo *REQUEST* no padrão FIPA-ACL para o agente responsável pela vaga (operação II).

Essa mensagem pode ser de dois tipos: *busy* indicando que um carro acabou de estacionar na vaga e *free* indicando que o carro saiu da vaga. O agente responsável pela vaga então irá identificar o tipo da mensagem e, com base nela, na operação III irá fazer uma chamada HTTP para a API, a qual irá atualizar o status da vaga no banco de dados.

Após receber a resposta da API, o agente Vaga envia uma resposta para o agente gerente através do protocolo FIPA-ACL na operação II, que recebe essa mensagem e publica no tópico MQTT da vaga a mensagem *green* para acender o *led* verde ou *red* para acender o *led* vermelho.

Para realizar a codificação foi utilizado a IDE do NetBeans, a linguagem Java e o framework para desenvolvimento de agentes JADE. Os códigos a seguir são do Raspberry Pi responsável por todas as vagas do setor A de um estacionamento com

duas vagas.

A Figura 32 ilustra o agente gerente sendo criado e iniciado.

Figura 32 – Criação e execução do agente gerente

```
1  controller = contController.createNewAgent("AgentPi", "agents.Manager", null);
2  controller.start();
```

Fonte: Autoria própria (2021)

A Figura 33 ilustra os agentes vagas sendo criados e iniciados. Nesse caso foram criados dois agentes, A1 e A2, que serão responsáveis pelas vagas A1 e A2 do estacionamento.

Figura 33 – Criação e execução dos agentes vagas

```
1  public static String sector = "A";
2  public static int nSpots = 2;
3
4  for(int i = 0; i < nSpots; i++) {
5      String agentName = sector + (i + 1);
6      controller = contController.createNewAgent(agentName, "agents.Spot", null);
7      controller.start();
8  }
```

Fonte: Autoria própria (2021)

Dentro do agente gerente, tem-se as variáveis de configuração do MQTT no Java, que é quem irá lidar com o recebimento das mensagens do *broker*.

Nesse momento, o agente gerente não tem conhecimento dos outros agentes (vagas) e também não possui a conexão com o MQTT ainda. A Figura 34 ilustra a vinculação dos dois comportamentos do agente gerente, *AcknowledgeSpots* e *Mqtt*.

Figura 34 – Vinculação dos comportamentos do agente gerente

```
1  spots = new ArrayList<>();
2  agentSpots = new ArrayList<>();
3  Mqtt mqtt = new Mqtt(spots, agentSpots);
4
5  addBehaviour(new AcknowledgeSpots(spots, agentSpots));
6  addBehaviour(mqtt);
```

Fonte: Autoria própria (2021)

Na linha 1 é definido uma lista de agentes e na linha 2 uma lista contendo o nome de cada um dos agentes que serão preenchidos assim que o comportamento

AcknowledgeSpots adicionado na linha 5 ser executado. E na linha 3, a instância do *callback* MQTT é criada e adicionado também como um comportamento do agente na linha 6.

Agora, na função *setup* do agente gerente a configuração do MQTT é executada (Figura 35) e a inscrição nos tópicos das vagas são feitos (Figura 36).

Figura 35 – Execução da configuração do MQTT

```
1  /**
2  * Start MQTT connection
3  */
4  client = new MqttClient(broker, getAID().getName());
5  client.setCallback(mqtt);
6  client.connect();
7  message.setPayload((getAID().getName() + " connected!").getBytes());
8  /**
9  * End MQTT connection
10 */
```

Fonte: Autoria própria (2021)

Aqui, na linha 5, a mesma instância do Mqtt que foi definida como um comportamento do agente é definida como o *callback* do MQTT que irá ser executado toda vez que um carro entrar/sair do estacionamento.

Figura 36 – Inscrição do agente gerente nos tópicos das vagas no *broker*

```
1  for(int i = 0; i < nSpots; i++) {
2      topics[i] = sector + (i + 1) + "/message";
3      client.publish(topics[i], message);
4
5      System.out.println("Message '" + message + "' sent to '" + topics[i] + "'\n");
6
7      client.subscribe(sector + (i + 1) + "/parked");
8
9      System.out.println(getAID().getName() + " subscribed to '" + sector + (i + 1) + "/parked'\n");
10 }
```

Fonte: Autoria própria (2021)

O agente gerente irá se inscrever nos tópicos *A1/parked* e *A2/parked* e publicará uma mensagem de conexão nos tópicos *A1/message* e *A2/message*.

A Figura 37 ilustra o comportamento *AcknowledgeSpots* do tipo *OneShotBehaviour* que servirá para o agente gerente conhecer todos os outros agentes vagas e é executado após inicializar os agentes.

Figura 37 – Comportamento *AcknowledgeSpots*

```java
DFAgentDescription template = new DFAgentDescription();
ServiceDescription sd = new ServiceDescription();
sd.setType("AcknowledgeSpots");
template.addServices(sd);

try {
    DFAgentDescription[] search = DFService.search(myAgent, template);

    for (int i = 0; i < search.length; i++) {
        AID agentName = search[i].getName();
        spots.add(agentName);
        agentSpots.add(agentName.toString().split(" ")[3].split("@")[0]);
    }

    System.out.println("Agente " + myAgent.getAID().getName() + " já conhece as vagas.\nTotal: " + spots.size());
    System.out.println("\n\n");

} catch (FIPAException e) {
    System.out.println("AcknowledgeSpots Exception: " + e.getMessage());
}
```

Fonte: Autoria própria (2021)

Na linha 7, é realizado uma busca geral dos agentes registrados no *Diretory Facilitator* e então, nas linhas 9 a 13, é iterado por cada elemento buscado para popular as listas criadas na Figura 34 com os dados dos agentes vagas.

Nesse momento, o agente gerente já conhece os agentes vagas disponíveis e já pode trocar mensagens com eles por meio do padrão FIPA-ACL. A Figura 38 ilustra o comportamento *Mqtt* do tipo *SimpleBehaviour* que servirá para o agente gerente receber as mensagens do ESP-12e e repassar a mensagem recebida para o agente vaga correspondente.

Figura 38 – Comportamento *Mqtt*

```java
public void messageArrived(String topic, MqttMessage message) throws Exception {
    System.out.println("### MESSAGE ARRIVED ###\n");

    String messageString = new String(message.getPayload());
    System.out.println("Message '" + messageString + "' received from '" + topic + "'\n");

    String spotName = topic.split("/")[0];

    Integer index = agentSpots.indexOf(spotName);

    ACLMessage contrato = new ACLMessage(ACLMessage.REQUEST);
    contrato.addReceiver(spots.get(index));
    contrato.setContent(messageString);
    contrato.setConversationId("spot-update");

    System.out.println("### SENDING MESSAGE TO AGENT " + spotName + " ###\n");

    myAgent.send(contrato);
}
```

Fonte: Autoria própria (2021)

Basicamente, o método *messageArrived* irá ser executado toda vez que uma

mensagem chegar nos tópicos em que o agente gerente está inscrito. Então, ao chegar uma mensagem, na linha 9 será feito uma busca pelo nome do agente que é responsável pela vaga em que um carro chegou/saiu. Assim, na linha 11 uma mensagem no padrão FIPA-ACL é criado, na linha 12 este agente vaga é setado como o recebedor da mensagem e na linha 13, o conteúdo da mensagem recebida é setada como o conteúdo da mensagem e na linha 18 a mensagem é enviada ao agente vaga.

Os agentes vagas possuirão apenas um comportamento, *UpdateSpot*, um *CyclicBehaviour* que será responsável por receber as mensagens do agente gerente e então atualizar a vaga correspondente na API. A Figura 39 ilustra a configuração de um agente vaga.

Figura 39 – Configuração de um agente vaga

```
1  @Override
2  protected void setup(){
3      System.out.println("Iniciando agente Spot: " + getAID().getName());
4      addBehaviour(new UpdateSpot());
5  }
```

Fonte: Autoria própria (2021)

A Figura 40 ilustra o método *action* do comportamento *UpdateSpot* de cada agente vaga.

Figura 40 – Comportamento *UpdateSpot*

```
1  public void action() {
2      MessageTemplate contrato = MessageTemplate.MatchPerformative(ACLMessage.REQUEST);
3
4      ACLMessage msg = myAgent.receive(contrato);
5      String agentName = myAgent.getAID().getName().split("@")[0];
6
7      if(msg != null){
8          System.out.println("### AGENT " + agentName + " RECEIVED MESSAGE ###\n");
9          String messageContent = msg.getContent();
10         String topic = agentName + "/message";
11
12         Request request = new Request(agentName);
13
14         if(messageContent.equals("busy")) {
15             publishes(topic, "red");
16             System.out.println("### " + agentName + " UPDATED TO BUSY ###\n");
17             request.carArrived();
18         } else if(messageContent.equals("free")) {
19             publishes(topic, "green");
20             System.out.println("### " + agentName + " UPDATED TO FREE ###\n");
21             request.carLeft();
22         } else {
23             System.out.println("Waiting...\n");
24         }
25     }
26 }
```

Fonte: Autoria própria (2021)

Por ser um *CyclicBehaviour*, o comportamento é cíclico e fica executando por várias vezes seguidas. Então, da linhas 2 até a linha 4, uma mensagem é instanciada e na linha 7 é verificado se há um valor dentro da mensagem. Se não houver, o método *action* irá executar novamente. Se houver um valor na mensagem então quer dizer que um carro entrou/saiu do estacionamento e é necessário atualizar o status da vaga no banco de dados.

Na linha 12, uma instância da classe que faz requisições ao banco é criada e o nome do agente é passado como parâmetro para ela. Assim, a vaga em que o status será atualizado no banco é a mesma vaga que recebeu a mensagem do agente gerente.

Na linha 14 é verificado se o valor da mensagem é igual a *busy*, significa que um carro chegou no estacionamento. Nesse momento, o agente vaga publica no tópico MQTT (Figura 41) a mensagem *red*, indicando que o ESP-12e deve ligar o *led* vermelho. Por fim, a requisição para atualizar o status da vaga para ocupado é feita (Figura 42).

Já na linha 15 é verificado se o valor da mensagem é igual a *free*, significa que um carro saiu do estacionamento. Nesse momento, o agente vaga publica no tópico MQTT (Figura 41) a mensagem *green*, indicando que o ESP-12e deve ligar o *led* verde. Por fim, a requisição para atualizar o status da vaga para ocupado é feita (Figura 43).

Figura 41 – Método que publica no tópico do *broker*

```
1  public void publishes(String topic, String mensagem) {
2      message.setPayload(mensagem.getBytes());
3
4      try {
5          client.publish(topic, message);
6          System.out.println("Message '" + message + "' sent to '" + topic + "'\n");
7      } catch (MqttException e) {
8          e.printStackTrace();
9      }
10 }
```

Fonte: Autoria própria (2021)

Figura 42 – Método que atualiza uma vaga para ocupado na API

```
1  public static void carArrived() {
2      URL url = new URL("https://lfdczpirjbhn5hx2teevb2pobi.appsync-api.us-east-1.amazonaws.com/graphql");
3      HttpURLConnection conn = (HttpURLConnection) url.openConnection();
4      conn.setDoOutput(true);
5      conn.setRequestMethod("GET");
6      conn.setRequestProperty("Content-Type", "application/json");
7
8
9      String input="{\"query\":\"mutation{carArrived(spotId: \\\"" + spot + "\\\"){date driverId price endTime spot startTime status}}\\n\"}";
10 }
```

Fonte: Autoria própria (2021)

Figura 43 – Método que atualiza uma vaga para livre na API

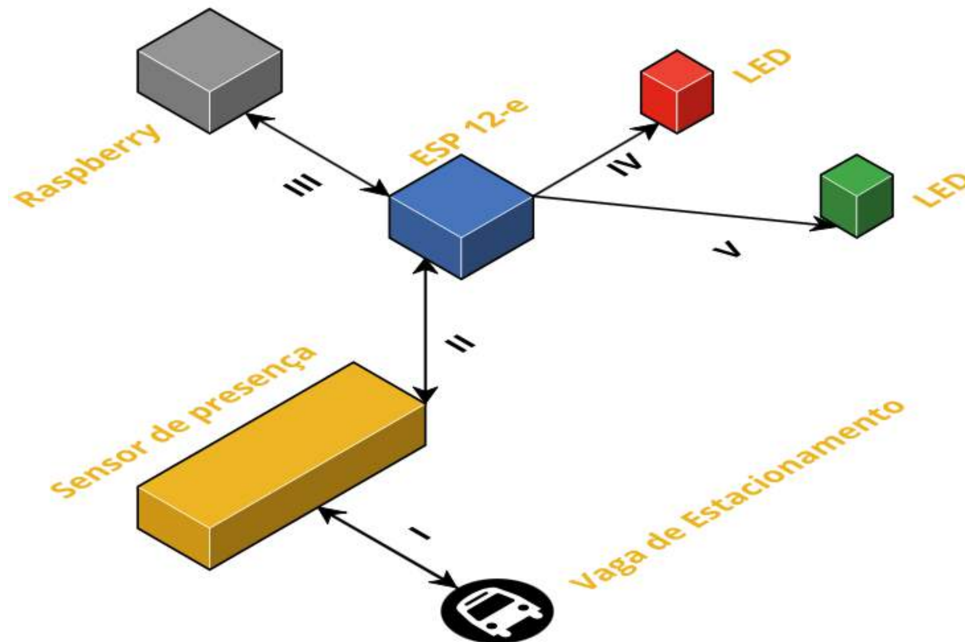
```
1  public static void carLeft() {
2      URL url = new URL("https://lfdczpirjbhn5hx2teevb2pobi.appsync-api.us-east-1.amazonaws.com/graphql");
3      HttpURLConnection conn = (HttpURLConnection) url.openConnection();
4      conn.setDoOutput(true);
5      conn.setRequestMethod("GET");
6      conn.setRequestProperty("Content-Type", "application/json");
7
8
9      String input="{\"query\":\"mutation{carLeft(spotId: \\\"" + spot + "\\\"){date driverId price endTime spot startTime status}}\\n\"}";
10 }
```

Fonte: Autoria própria (2021)

4.4.3 ESP-12e

Cada módulo ESP-12e e seus componentes representam uma vaga do estacionamento. A Figura 44 apresenta o fluxo de comunicação entre os dispositivos físicos conectados ao módulo ESP-12e. Para isso, o diagrama demonstra a detecção de presença do carro na vaga, o recebimento de uma mensagem através do Raspberry até o momento em que o *led* correspondente é aceso.

Figura 44 – Diagrama do módulo ESP-12e



Fonte: Autoria própria (2021)

O detector de presença e os *leds* vermelho e verde estão conectados ao ESP-12e por meio de fios ligados aos pinos do módulo, ilustrados nas operações II, IV e V. Assim, o ESP-12e é capaz de receber as distâncias calculadas pelo detector de presença e passar energia para os *leds* acenderem.

A distância no detector de presença deve ser ajustada para que haja uma distância padrão, por exemplo, entre o sensor e a parede, a que possa ser comparada com a distância atual. Deve-se também manter a distância anterior para evitar o envio mensagens repetidas para o *broker*. Portanto, quando um carro entra/sai da vaga, apenas uma mensagem é enviada, e isso é apenas possível comparando a distância anterior com a distância atual.

Na operação I, se distância atual é menor que a distância padrão significa que um carro está estacionado na vaga. Então, o ESP-12e, representando uma vaga do estacionamento, publica no tópico MQTT correspondente a sua vaga que um carro foi estacionado e então, o processo descrito na subseção 4.4.2 ocorre. O mesmo acontece se a distância atual for igual a distância padrão, significa que a vaga está livre.

Assim que o Raspberry envia novamente a mensagem de retorno para o ESP-12e (operação III), ele então identifica a mensagem *green* ou *red* e acende o *led* correspondente através das operações IV e V.

Para realizar a codificação foi utilizado a IDE do Arduino e a linguagem C++.

Os códigos a seguir são do ESP-12e responsável pela vaga 1 do setor A de um estacionamento.

Para realizar a comunicação com o *broker*, o ESP-12e precisa estar conectado à Internet, para isso foi necessário utilizar duas bibliotecas, a *ESP8266WiFi* para realizar a conexão wifi e a *PubSubClient* para realizar a conexão com o MQTT.

A Figura 45 ilustra a função *setup* que é executada toda vez que é ligado ou reiniciado.

Figura 45 – Função *setup*

```
1  void setup() {
2    pinMode(redLed, OUTPUT);
3    pinMode(greenLed, OUTPUT);
4    pinMode(trigPin, OUTPUT);
5    pinMode(echoPin, INPUT);
6
7    Serial.begin(9600);
8
9    connectWifi();
10   // connect to MQTT server
11   client.setServer(serverHostname, 1883);
12   client.setCallback(callback);
13
14   Serial.println("Waiting to set up standard distance ...");
15   standardDistance = getDistance();
16
17   Serial.print("Standard distance setted up to: ");
18   Serial.println(standardDistance);
19 }
```

Fonte: Autoria própria (2021)

Nas linhas 2 a 5 é realizada a identificação de entrada e saída dos pinos, os pinos dos *leds* e o pino de *trigger* do sensor ultrassônico são definidos como pinos de saída nas portas D2, D1 e D4 e o pino de *echo* é definido como pino de entrada na porta D3. Nas linhas 9 a 12 é feita a conexão do ESP-12e com o wifi e a conexão com o *broker* MQTT, configurando o *callback* chamado toda vez que uma mensagem chegar aos tópicos inscritos. Na linha 15 a distância padrão é definida. O *callback* do MQTT e o cálculo da distância serão explicados nos parágrafos seguintes.

A Figura 46 ilustra como a distância é calculada pelo sensor ultrassônico.

Figura 46 – Função para o cálculo da distância

```
1  int getDistance(){
2    digitalWrite(trigPin, LOW);
3    delayMicroseconds(2);
4    digitalWrite(trigPin, HIGH);
5    delayMicroseconds(10);
6    digitalWrite(trigPin, LOW);
7
8    duration = pulseIn(echoPin, HIGH);
9    currentDistance = duration*0.034/2;
10
11   return currentDistance;
12 }
```

Fonte: Autoria própria (2021)

A linha 2 seta o pino de *trigger* como *LOW*, o que quer dizer que não há nenhuma tensão no pino. Após 2 microssegundos, na linha 3 é setado *HIGH* no pino, ou seja, uma tensão é adicionada e um pulso é emitido e então, após 2 microssegundos, a tensão é removida do pino. Na linha 8, a função *pulseIn* irá capturar, no pino *echo*, a duração do pulso que foi emitido pelo pino *trigger*. Assim, na linha 9 é possível calcular a distância em centímetros com base na equação 1.

A Figura 47 ilustra a função de *callback* que irá ser executada todas as vezes que o Raspberry publicar no tópico *spot1/message* em que o ESP-12e está inscrito.

Figura 47 – Função de *callback* do MQTT

```
1  void callback(char *msgTopic, byte *msgPayload, unsigned int msgLength) {
2    static char message[MAX_MSG_LEN+1];
3
4    if (msgLength > MAX_MSG_LEN) {
5      msgLength = MAX_MSG_LEN;
6    }
7
8    strncpy(message, (char *)msgPayload, msgLength);
9    message[msgLength] = '\0';
10
11   Serial.printf("message '%s' received from '%s'\n", message, topicMessage);
12   if (strcmp(message, "green") == 0) {
13     turnOnGreen();
14   } else if (strcmp(message, "red") == 0) {
15     turnOnRed();
16   } else {
17     Serial.println("Waiting ... ");
18   }
19 }
```

Fonte: Autoria própria (2021)

Na linha 2, são alocados 128 caracteres para a mensagem recebida. Assim, nas linhas 8 e 9 são atribuídos os valores recebidos no *payload* da mensagem via MQTT para a nova variável de mensagem. E assim, com base no valor da mensagem, *green* ou *red*, o *led* correspondente é aceso. A Figura 48 ilustra como as funções para acender os *leds* funcionam.

Figura 48 – Função para acender os *leds*

```
1  void turnOnRed() {
2    digitalWrite(redLed, HIGH);
3    digitalWrite(greenLed, LOW);
4  }
5
6  void turnOnGreen() {
7    digitalWrite(redLed, LOW);
8    digitalWrite(greenLed, HIGH);
9  }
```

Fonte: Autoria própria (2021)

Na função *turnOnRed*, a tensão é passada para o pino em que o *led* vermelho se encontra e removido tensão do pino em que o *led* verde se encontra. Já a função *turnOnGreen* faz o inverso.

A Figura 49 ilustra a função *loop* que é executada até que o ESP-12e seja desligado ou reiniciado.

Figura 49 – Função *loop*

```
void loop() {
  if (!client.connected()) {
    connectMQTT();
    client.subscribe(topicMessage);
  }

  client.loop();

  currentDistance = getDistance();

  Serial.print("Current distance: ");
  Serial.println(currentDistance);

  if(currentDistance < (standardDistance - 2) && currentDistance != lastDistance){
    messageSent = "busy";
    client.publish(topicParked, messageSent.c_str());
    Serial.printf("message '%s' sent to '%s'\n", messageSent.c_str(), topicMessage);
  } else if (lastDistance != currentDistance) {
    messageSent = "free";
    client.publish(topicParked, messageSent.c_str());
    Serial.printf("message '%s' sent to '%s'\n", messageSent.c_str(), topicMessage);
  }

  lastDistance = currentDistance;

  if (currentDistance == standardDistance){
    Serial.println("Waiting for a car ...");
  }

  delay(5000);
}
```

Fonte: Autoria própria (2021)

A linha 2 irá sempre verificar se a conexão com o MQTT ainda está ativa, no caso da conexão ser perdida no meio do funcionamento. Assim, caso seja perdida, a conexão é feita novamente para que não haja problemas na comunicação.

A linha 9 irá ler a distância atual através do sensor ultrassônico (Figura 46). Assim, pode-se iniciar as comparações com a distância padrão para ver se mudou e assim tomar conhecimento se um carro entrou ou saiu do estacionamento.

Nesse caso, é levado em consideração uma taxa de erro de 2 centímetros para a distância. Então, se a distância atual for menor que a distância padrão menos 2 centímetros e a distância atual não é a última distância lida, quer dizer que um carro estacionou na vaga, então a mensagem *busy* é publicada no tópico *spot1/parked* (linha 16). Caso contrário, porém, ainda checando se a distância atual não é a última distância lida, quer dizer que o carro saiu do estacionamento, então a mensagem *free* é publicada no tópico *spot1/parked* (linha 20). Por último, na linha 30 é setado um *delay* de 5000 milissegundos para que a próxima execução do *loop* seja feita.

A Figura 50 ilustra como é feita a conexão do ESP-12e ao *broker* MQTT.

Figura 50 – Função de conexão ao *broker* MQTT

```
1  void connectMQTT() {
2    while (!client.connected()) {
3      String clientId = "Spot1";
4      Serial.printf("MQTT connecting as client %s ... \n", clientId.c_str());
5
6      if (client.connect(clientId.c_str())) {
7        Serial.println("MQTT connected");
8
9        messageSent = clientId + "connected!";
10       client.publish(topicMessage, messageSent.c_str());
11       Serial.printf("message '%s' sent to '%s'\n", messageSent.c_str(), topicMessage);
12
13       client.subscribe(topicMessage);
14
15     } else {
16       Serial.printf("MQTT failed, state %s, retrying ... \n", client.state());
17       delay(2500);
18     }
19   }
20 }
21
```

Fonte: Autoria própria (2021)

Na linha 6 uma tentativa de conexão com o *broker* é feita, em caso de sucesso, uma mensagem de conexão será enviada ao tópico *spot1/message*, caso a tentativa de conexão falhe, é realizada uma nova tentativa até que a conexão seja realizada.

4.5 Considerações finais

Este capítulo apresentou o desenvolvimento do protótipo ciber-físico e suas integrações. Foram apresentadas as funcionalidades do Sistema Ciber-Físico com os agentes dentro do Raspberry, e como foi feita a implementação utilizando o JADE. Também foram apresentadas as funcionalidades do sistema de gerenciamento e como foi realizada a integração dos componentes por meio da API.

O próximo capítulo apresenta os cenários criados para testar o protótipo implementado e os resultados que a arquitetura apresentou com relação a: comunicação entre os componentes; sistema em tempo real; e o número de vagas possíveis a serem alocadas por um único Raspberry.

5 RESULTADOS

Neste capítulo são apresentados os resultados obtidos por meio da execução de cenários de testes do protótipo desenvolvido em um ambiente próximos a realidade de um estacionamento inteligente. Estes cenários foram desenvolvidos com o intuito de validar o funcionamento e aplicabilidade do sistema ciber-físico e a comunicação com o motorista e o administrador por meio do sistema desenvolvido.

É importante ressaltar que, o objetivo deste capítulo é analisar somente o funcionamento do sistema ciber-físico e a comunicação com o sistema em nuvem. Portanto, não são abordadas características como a negociação das vagas, abrir e fechar cancelas do estacionamento e a maneira como é verificado se um motorista estacionou em uma vaga. Assume-se que todos os motoristas obedecem rigorosamente e estacionam exatamente na vaga em que foi solicitada.

5.1 Cenários de teste

Para implementar os cenários de testes é necessário configurar o ambiente do sistema ciber-físico. Todos os experimentos foram executados utilizando os seguintes hardwares:

- 1 Raspberry Pi Model B;
- 2 ESP-12e;
- 2 Sensores ultrassônico HC-SR04;
- 2 LEDs difuso 5mm verde;
- 2 LEDs difuso 5mm vermelho;
- 2 Protoboards;

Para a utilização do sistema de gerenciamento do estacionamento, é necessário configurar o ambiente React. Para isso, os experimentos realizados utilizaram as seguintes configurações:

- macOS Big Sur versão 11.5.2;
- Node versão 14.17.1;
- Yarn versão 1.22.10;

A seguir são apresentados os experimentos realizados:

- Experimento 1: Análise da comunicação dos componentes com dez vagas ao total e dois motoristas disputando duas vagas livres;

- Experimento 2: Análise da comunicação em tempo real com dez vagas ao total e dois motoristas disputando uma vaga livre;
- Experimento 3: Análise da quantidade de vagas possíveis utilizando um Raspberry para controlar as vagas do estacionamento.

5.2 Configuração dos experimentos

Para realizar os experimentos 1 e 2 é necessário que os componentes da arquitetura ciber-física estejam ligados na tomada e devidamente conectados a Internet e que o sistema de gerenciamento esteja sendo executado. Ainda, é necessário que os agentes estejam criados e o agente gerente conectado ao *broker* MQTT, do mesmo modo, os módulos ESP-12e também devem estar conectados ao *broker*.

A Figura 51 ilustra o agente gerente e os agentes vagas sendo instanciados no Raspberry.

Figura 51 – Agentes criados no Raspberry

```
INFO: --------------------------------------
Agent container Main-Container@192.168.0.104 is ready.
--------------------------------------------
Iniciando agente Manager: AgentPi@192.168.0.104:1099/JADE
Iniciando agente Spot: A1@192.168.0.104:1099/JADE
Iniciando agente Spot: A2@192.168.0.104:1099/JADE
Iniciando agente Spot: A3@192.168.0.104:1099/JADE
Iniciando agente Spot: A4@192.168.0.104:1099/JADE
Iniciando agente Spot: A5@192.168.0.104:1099/JADE
Iniciando agente Spot: A6@192.168.0.104:1099/JADE
Iniciando agente Spot: A7@192.168.0.104:1099/JADE
Iniciando agente Spot: A8@192.168.0.104:1099/JADE
Iniciando agente Spot: A9@192.168.0.104:1099/JADE
Iniciando agente Spot: A10@192.168.0.104:1099/JADE
```

Fonte: Autoria própria (2021)

A Figura 52 ilustra o *broker* MQTT funcionando com o agente gerente conectado como *AgentPi* e dois ESP-12e conectados como as vagas A1 e A2.

Figura 52 – *Broker* com dispositivos conectados

```
pi@raspberrypi:~ $ mosquitto
1635968697: mosquitto version 1.5.7 starting
1635968697: Using default config.
1635968697: Opening ipv4 listen socket on port 1883.
1635968697: Opening ipv6 listen socket on port 1883.
1635969103: New connection from 192.168.0.105 on port 1883.
1635969104: New client connected from 192.168.0.105 as AgentPi@192.168.0.105:1099/JADE (c1, k60).
1635969164: New connection from 192.168.0.106 on port 1883.
1635969164: New client connected from 192.168.0.106 as A1 (c1, k15).
1635969251: New connection from 192.168.0.107 on port 1883.
1635969251: New client connected from 192.168.0.107 as A2 (c1, k15).
```

Fonte: Autoria própria (2021)

A Figura 53 ilustra o agente gerente se inscrevendo nos tópicos MQTT de cada vaga e a execução do comportamento para o agente gerente conhecer todos os agentes vagas.

Figura 53 – SMA e *broker* MQTT rodando no Raspberry

```
Message 'AgentPi@192.168.0.104:1099/JADE connected!' sent to 'A1/message'

AgentPi@192.168.0.104:1099/JADE subscribed to 'A1/parked'

Message 'AgentPi@192.168.0.104:1099/JADE connected!' sent to 'A2/message'

AgentPi@192.168.0.104:1099/JADE subscribed to 'A2/parked'

Message 'AgentPi@192.168.0.104:1099/JADE connected!' sent to 'A3/message'

AgentPi@192.168.0.104:1099/JADE subscribed to 'A3/parked'

Message 'AgentPi@192.168.0.104:1099/JADE connected!' sent to 'A4/message'

AgentPi@192.168.0.104:1099/JADE subscribed to 'A4/parked'

Message 'AgentPi@192.168.0.104:1099/JADE connected!' sent to 'A5/message'

AgentPi@192.168.0.104:1099/JADE subscribed to 'A5/parked'

Message 'AgentPi@192.168.0.104:1099/JADE connected!' sent to 'A6/message'

AgentPi@192.168.0.104:1099/JADE subscribed to 'A6/parked'

Message 'AgentPi@192.168.0.104:1099/JADE connected!' sent to 'A7/message'

AgentPi@192.168.0.104:1099/JADE subscribed to 'A7/parked'

Message 'AgentPi@192.168.0.104:1099/JADE connected!' sent to 'A8/message'

AgentPi@192.168.0.104:1099/JADE subscribed to 'A8/parked'

Message 'AgentPi@192.168.0.104:1099/JADE connected!' sent to 'A9/message'

AgentPi@192.168.0.104:1099/JADE subscribed to 'A9/parked'

Message 'AgentPi@192.168.0.104:1099/JADE connected!' sent to 'A10/message'

AgentPi@192.168.0.104:1099/JADE subscribed to 'A10/parked'

Agente AgentPi@192.168.0.104:1099/JADE já conhece as vagas.
Total: 10
```

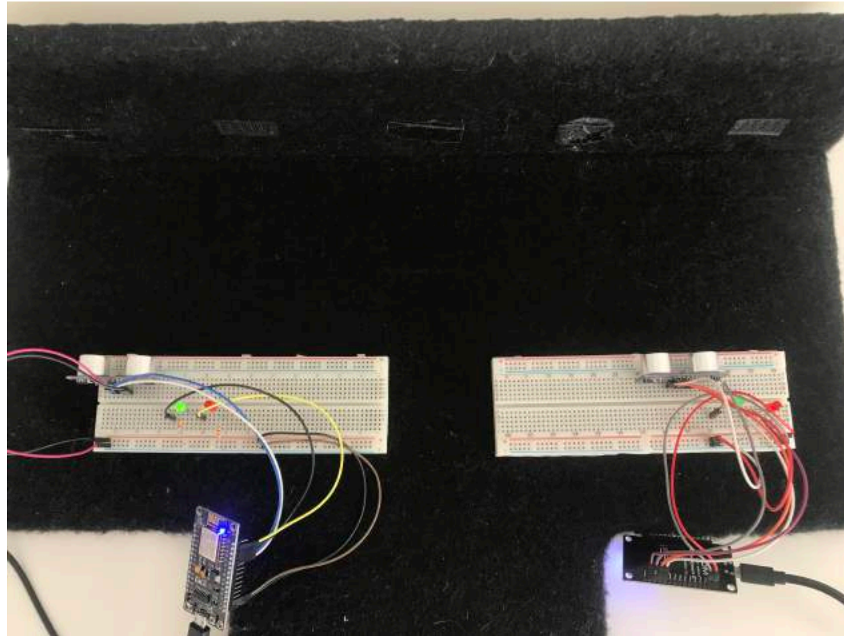
Fonte: Autoria própria (2021)

5.3 Experimento 1

O objeto do estudo desta seção é a comunicação entre todos os componentes do sistema. Para tanto, foi executado um cenário de teste em um setor do estacionamento com dez vagas no total, sendo duas vagas físicas e oito virtuais. Dessas dez vagas, oito estão ocupadas, duas estão livres e há dois motoristas disputando as duas vagas livres.

A Figura 54 ilustra as duas vagas livres em um setor do estacionamento.

Figura 54 – Duas vagas livres no estacionamento



Fonte: Autoria própria (2021)

A Figura 55 ilustra o estado do banco de dados com as dez vagas de um setor do estacionamento, onde há oito ocupadas e duas livres.

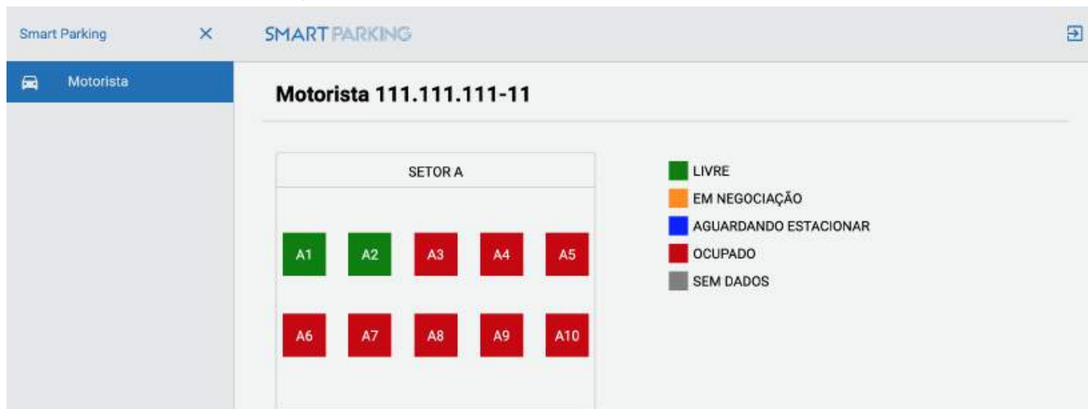
Figura 55 – Estado do banco de dados com duas vagas livres

spot	date	driverId	endTime	price	startTime	status
A1	null	null	null	null	null	FREE
A10	2021-11-03	101.101.101-10	15:30	10	14:30	BUSY
A2	null	null	null	null	null	FREE
A3	2021-11-03	333.333.333-33	15:28	10	14:28	BUSY
A4	2021-11-03	444.444.444-44	15:28	10	14:28	BUSY
A5	2021-11-03	555.555.555-55	15:29	10	14:29	BUSY
A6	2021-11-03	666.666.666-66	15:25	10	14:25	BUSY
A7	2021-11-03	777.777.777-77	15:28	10	14:28	BUSY
A8	2021-11-03	888.888.888-88	15:34	10	14:34	BUSY
A9	2021-11-03	999.999.999-99	15:31	10	14:31	BUSY

Fonte: Autoria própria (2021)

A Figura 56 ilustra o motorista 111.111.111-11 logado no sistema de gerenciamento do estacionamento visualizando as vagas disponíveis e não disponíveis.

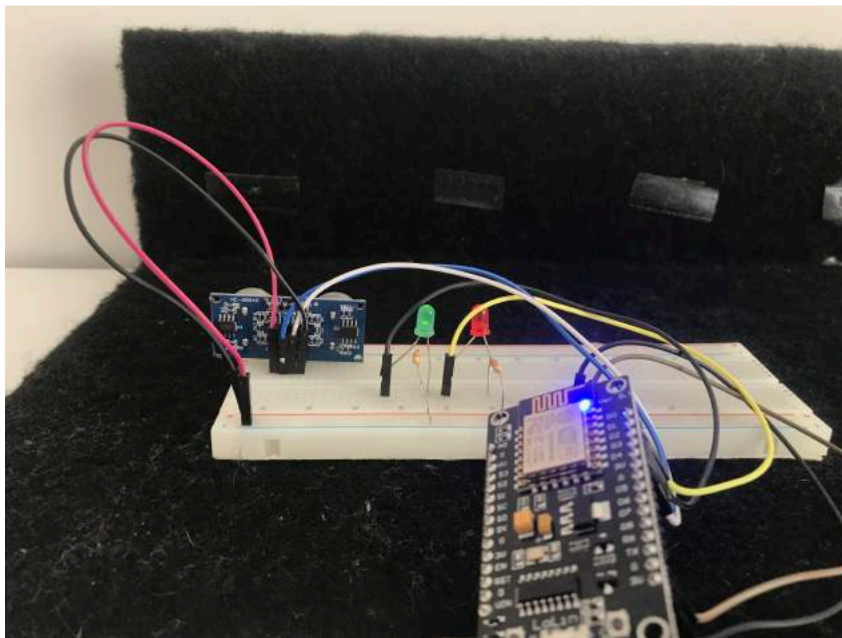
Figura 56 – Tela do Motorista 111.111.111-11



Fonte: Autoria própria (2021)

A Figura 57 ilustra a vaga A1, que está livre, do estacionamento.

Figura 57 – Vaga A1 no estacionamento



Fonte: Autoria própria (2021)

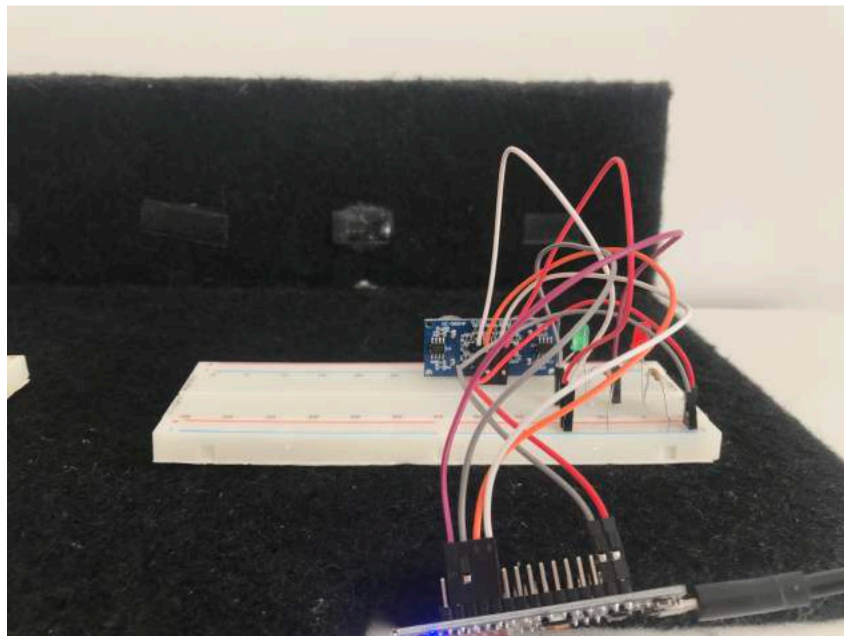
A Figura 58 ilustra o motorista 222.222.222-22 logado no sistema de gerenciamento do estacionamento visualizando as vagas disponíveis e não disponíveis.

Figura 58 – Tela do Motorista 222.222.222-22

Fonte: Autoria própria (2021)

A Figura 59 ilustra a vaga A2, que está livre, do estacionamento.

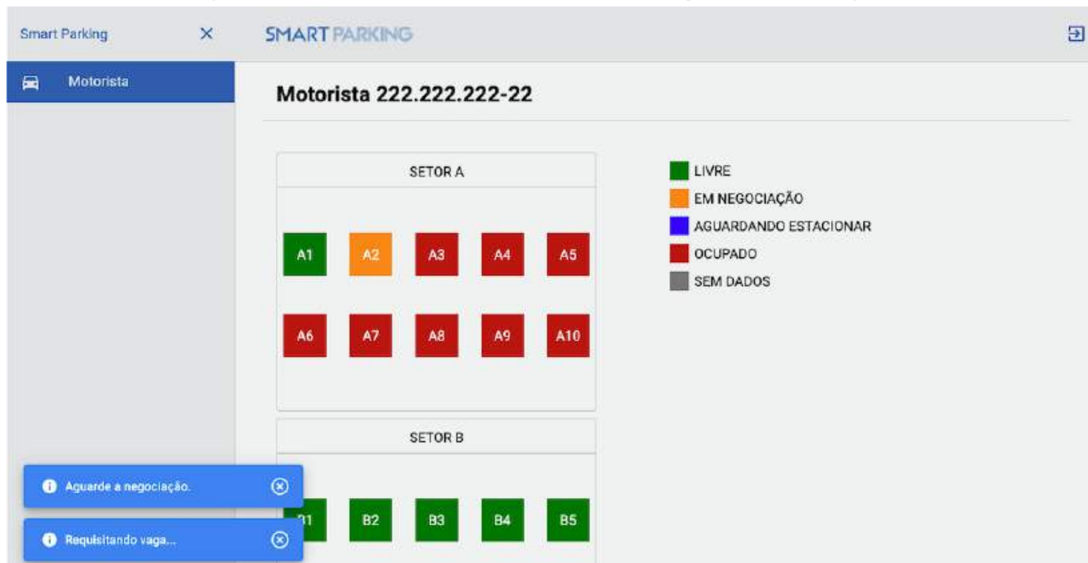
Figura 59 – Vaga A2 no estacionamento



Fonte: Autoria própria (2021)

A Figura 60 ilustra a vaga A2 sendo requisitada pelo motorista 222.222.222-22 e sendo processada pela negociação. Nessa figura, é possível ver que a vaga A2 que está sendo negociada mudou seu estado para "Em negociação".

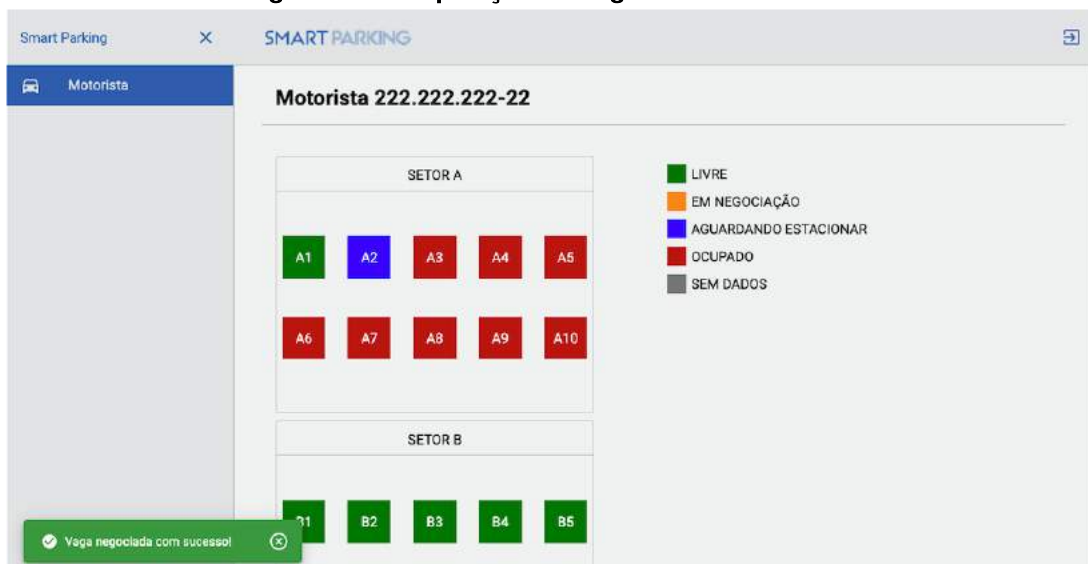
Figura 60 – Motorista 222.222.222-22 requisitando vaga A2



Fonte: Autoria própria (2021)

A Figura 61 ilustra a vaga A2 negociada e aguardando o motorista estacionar. Nessa figura, é possível ver que a vaga A2 que foi negociada mudou seu estado para "Aguardando estacionar".

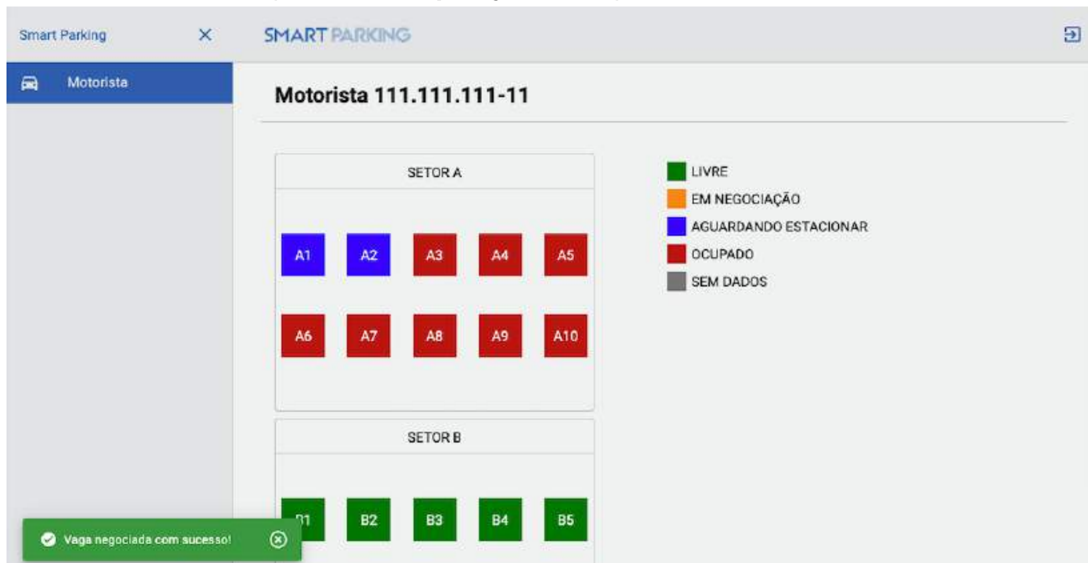
Figura 61 – Requisição da vaga A2 com sucesso



Fonte: Autoria própria (2021)

Do mesmo modo, o motorista 111.111.111-11 também requisitou a vaga A1. Na Figura 62 é possível visualizar as vagas A1 e A2 aguardando os motoristas estacionarem.

Figura 62 – Requisição da vaga A1 com sucesso



Fonte: Autoria própria (2021)

A Figura 63 ilustra o estado do banco de dados após as requisições feitas através do sistema de gerenciamento do estacionamento. As vagas A1 e A2 que antes estavam com o status *FREE*, agora estão com o status *WAITING_FOR_PARKING*.

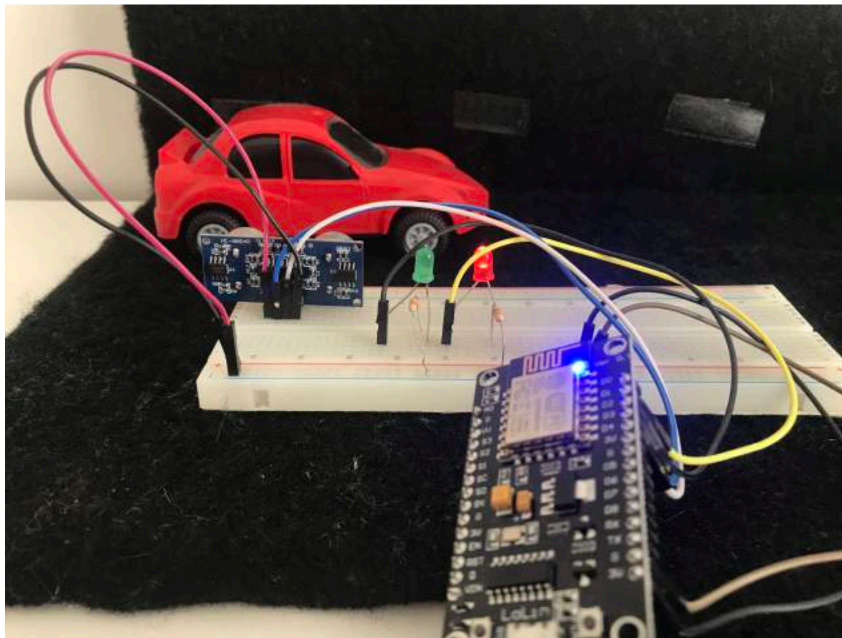
Figura 63 – Estado do banco de dados após requisições

spot	date	driverId	endTime	price	startTime	status
A1	2021-11-03	111.111.111-11	18:08	10	17:08	WAITING_FOR_PARKING
A10	2021-11-03	101.101.101-10	15:30	10	14:30	BUSY
A2	2021-11-03	222.222.222-22	18:06	10	17:06	WAITING_FOR_PARKING
A3	2021-11-03	333.333.333-33	15:28	10	14:28	BUSY
A4	2021-11-03	444.444.444-44	15:28	10	14:28	BUSY
A5	2021-11-03	555.555.555-55	15:29	10	14:29	BUSY
A6	2021-11-03	666.666.666-66	15:25	10	14:25	BUSY
A7	2021-11-03	777.777.777-77	15:28	10	14:28	BUSY
A8	2021-11-03	888.888.888-88	15:34	10	14:34	BUSY
A9	2021-11-03	999.999.999-99	15:31	10	14:31	BUSY

Fonte: Autoria própria (2021)

A Figura 64 ilustra o motorista estacionado na vaga A1 e o *led* vermelho aceso.

Figura 64 – Motorista estaciona na vaga A1



Fonte: Autoria própria (2021)

A Figura 65 mostra os *logs* de comunicação entre o ESP-12e e o agente gerente por meio do tópico *A1/parked* no *broker* e o agente gerente replicando a mensagem recebida para o agente responsável pela vaga A1.

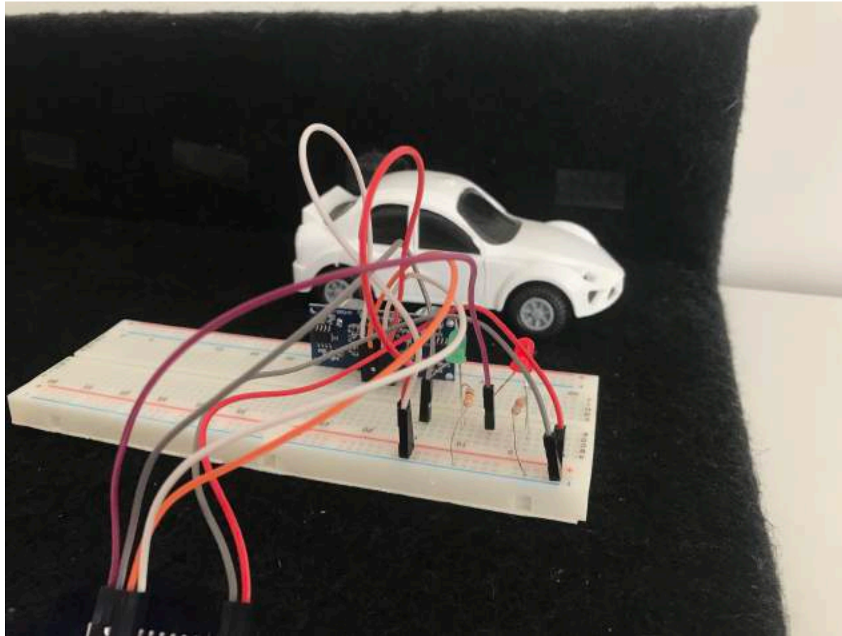
Figura 65 – Troca de mensagens entre os dispositivos e os agentes

```
### MESSAGE ARRIVED ###

Message 'busy' received from 'A1/parked'

### SENDING MESSAGE TO AGENT A1 ###

### AGENT A1 RECEIVED MESSAGE ###

Message 'red' sent to 'A1/message'

### A1 UPDATED TO BUSY ###
```

Fonte: Autoria própria (2021)

Do mesmo modo, a Figura 66 ilustra o motorista estacionado na vaga A2 e o *led* vermelho aceso.

Figura 66 – Motorista estaciona na vaga A2



Fonte: Autoria própria (2021)

E assim, a Figura 65 mostra os *logs* de comunicação entre o ESP-12e e o agente gerente por meio do tópico *A2/parked* no *broker* e o agente gerente replicando a mensagem recebida para o agente responsável pela vaga A2.

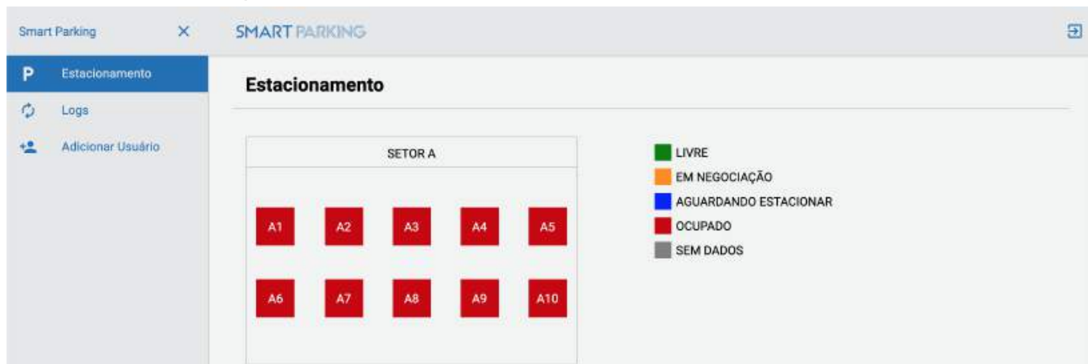
Figura 67 – Troca de mensagens entre os dispositivos e os agentes

```
### MESSAGE ARRIVED ###

Message 'busy' received from 'A2/parked'

### SENDING MESSAGE TO AGENT A2 ###

### AGENT A2 RECEIVED MESSAGE ###

Message 'red' sent to 'A2/message'

### A2 UPDATED TO BUSY ###
```

Fonte: Autoria própria (2021)

A partir do momento que o agente vaga envia ao ESP-12e uma mensagem para ligar o *led* vermelho, a requisição para a API também é feita, assim, atualizando em tempo real o sistema de gerenciamento do estacionamento. A Figura 68 ilustra a tela do administrador visualizando todas as vagas do setor A ocupadas.

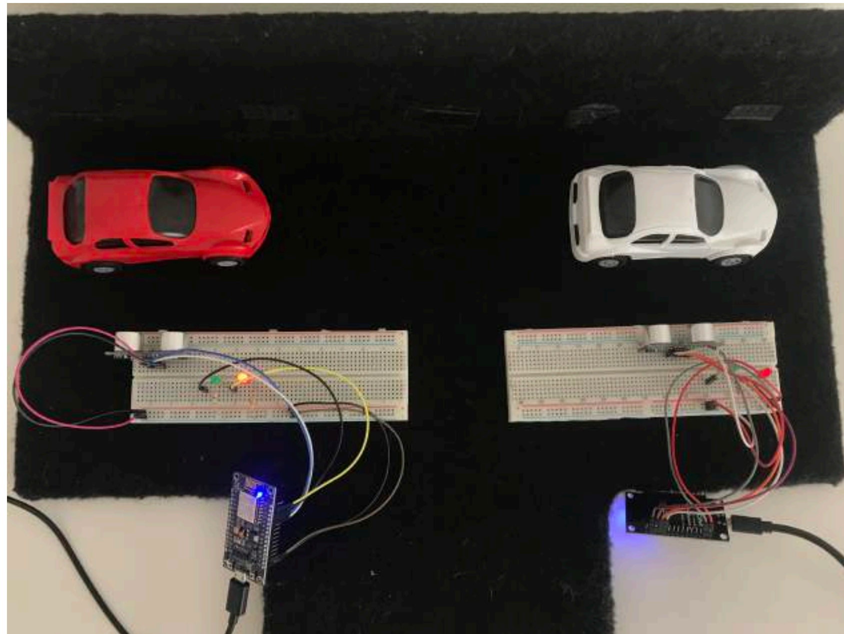
Figura 68 – Tela do administrador do estacionamento



Fonte: Autoria própria (2021)

A Figura 69 ilustra as duas vagas do estacionamento ocupadas.

Figura 69 – Duas vagas ocupadas no estacionamento



Fonte: Autoria própria (2021)

A Figura 70 ilustra o estado do banco de dados após os dois motoristas estacionarem nas vagas.

Figura 70 – Estado do banco de dados após estacionar

spot	date	driverId	endTime	price	startTime	status
A1	2021-11-03	111.111.111-11	15:38	10	14:38	BUSY
A10	2021-11-03	101.101.101-10	15:30	10	14:30	BUSY
A2	2021-11-03	222.222.222-22	15:41	10	14:41	BUSY
A3	2021-11-03	333.333.333-33	15:28	10	14:28	BUSY
A4	2021-11-03	444.444.444-44	15:28	10	14:28	BUSY
A5	2021-11-03	555.555.555-55	15:29	10	14:29	BUSY
A6	2021-11-03	666.666.666-66	15:25	10	14:25	BUSY
A7	2021-11-03	777.777.777-77	15:28	10	14:28	BUSY
A8	2021-11-03	888.888.888-88	15:34	10	14:34	BUSY
A9	2021-11-03	999.999.999-99	15:31	10	14:31	BUSY

Fonte: Autoria própria (2021)

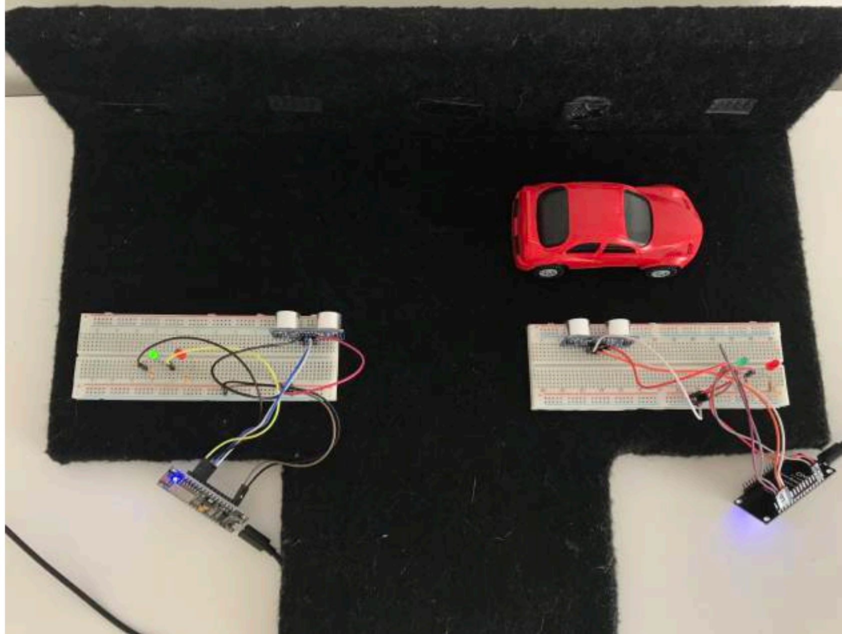
Com base neste experimento, é possível notar que a integração entre todos os componentes funcionou de forma que um estacionamento inteligente possa ser representado. O protocolo MQTT se mostrou eficiente no trabalho de comunicação dos dispositivos físicos, mesmo tendo apenas duas vagas físicas. Um outro ponto importante a se destacar neste experimento é a comunicação entre os agentes dentro do Raspberry, dividindo o trabalho de receber e mandar mensagens no *broker* e se comunicar com o sistema em nuvem.

5.4 Experimento 2

O objeto do estudo desta seção é validar o tempo real do sistema por meio das assinaturas para que não haja dois motoristas para estacionar na mesma vaga. Para isso, foi executado um cenário de teste em um setor do estacionamento com dez vagas no total, sendo duas vagas físicas e oito virtuais. Dessas dez vagas, nove estão ocupadas, uma está livre e há dois motoristas disputando a vaga livre.

A Figura 71 ilustra uma vaga livre em um setor do estacionamento.

Figura 71 – Uma vaga livre no estacionamento



Fonte: Autoria própria (2021)

A Figura 72 ilustra o estado do banco de dados com as dez vagas de um setor do estacionamento, onde há nove ocupadas e uma livre.

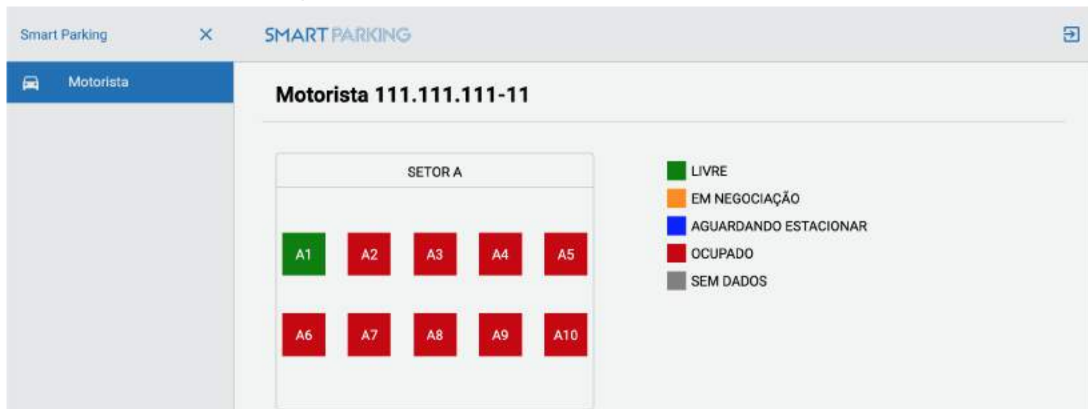
Figura 72 – Estado do banco de dados com uma vaga livre

spot	date	driverId	endTime	price	startTime	status
A1	null	null	null	null	null	FREE
A10	2021-11-03	101.101.101-10	15:30	10	14:30	BUSY
A2	2021-11-03	121.212.121-21	15:27	10	14:27	BUSY
A3	2021-11-03	333.333.333-33	15:28	10	14:28	BUSY
A4	2021-11-03	444.444.444-44	15:28	10	14:28	BUSY
A5	2021-11-03	555.555.555-55	15:29	10	14:29	BUSY
A6	2021-11-03	666.666.666-66	15:25	10	14:25	BUSY
A7	2021-11-03	777.777.777-77	15:28	10	14:28	BUSY
A8	2021-11-03	888.888.888-88	15:34	10	14:34	BUSY
A9	2021-11-03	999.999.999-99	15:31	10	14:31	BUSY

Fonte: Autoria própria (2021)

A Figura 73 ilustra o motorista 111.111.111-11 logado no sistema de gerenciamento do estacionamento visualizando a vaga disponível e as não disponíveis.

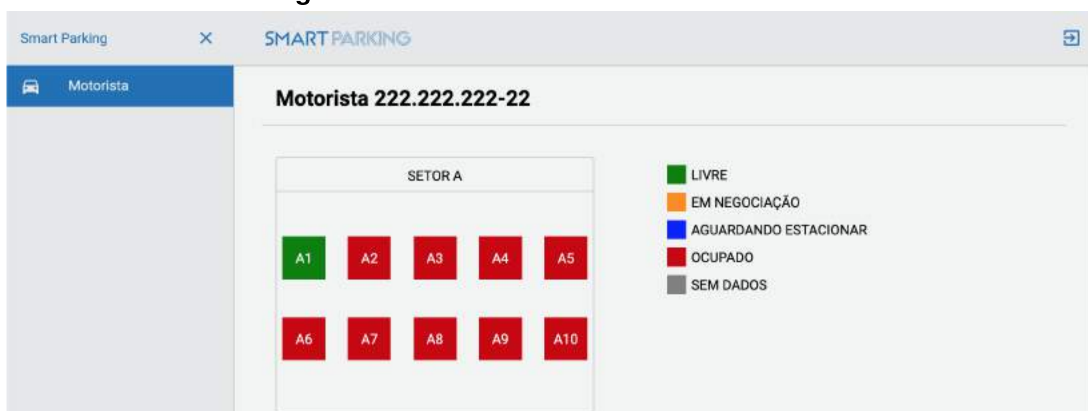
Figura 73 – Tela do Motorista 111.111.111-11



Fonte: Autoria própria (2021)

A Figura 74 ilustra o motorista 222.222.222-22, ao mesmo tempo, logado no sistema de gerenciamento do estacionamento visualizando a vaga disponível e as não disponíveis.

Figura 74 – Tela do Motorista 222.222.222-22



Fonte: Autoria própria (2021)

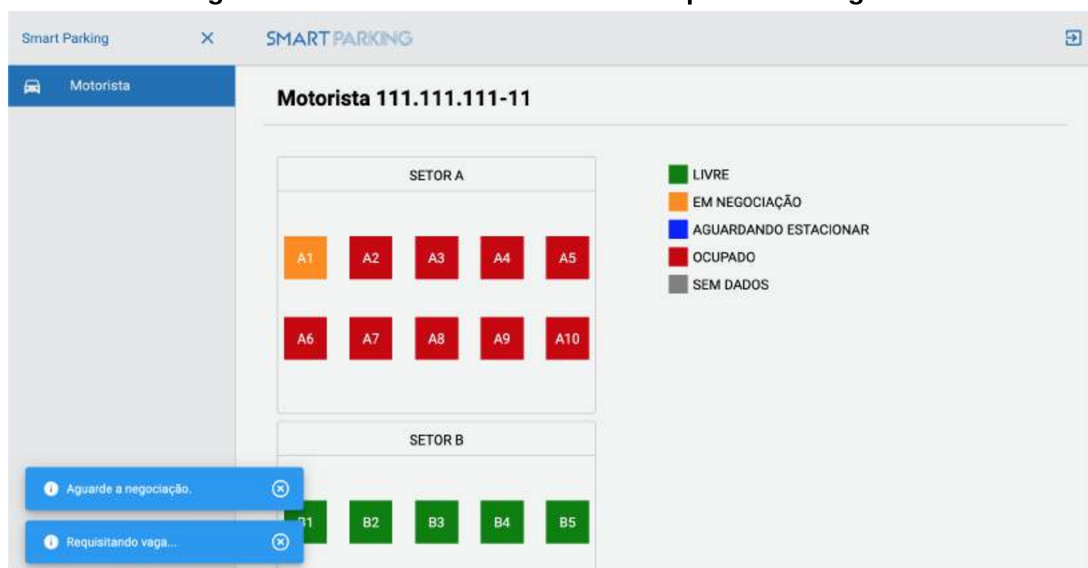
A Figura 75 ilustra a vaga A1, que está livre, do estacionamento.

Figura 75 – Vaga A1 no estacionamento

Fonte: Autoria própria (2021)

A Figura 76 ilustra a vaga A1 sendo requisitada pelo motorista 111.111.111-11 e sendo processada pela negociação. Nessa figura, é possível ver que a vaga A1 que está sendo negociada mudou seu estado para "Em negociação".

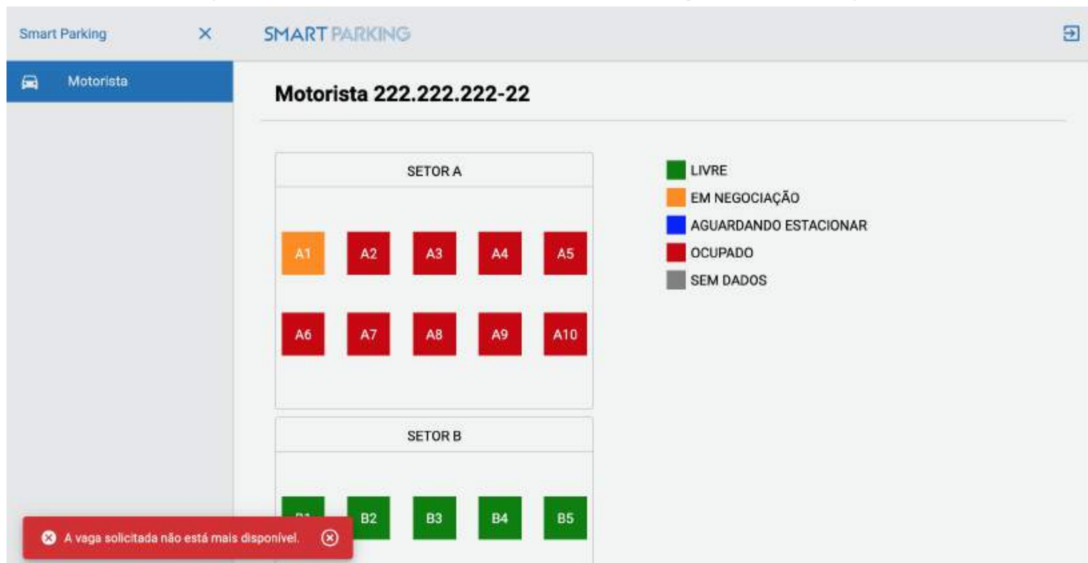
Figura 76 – Motorista 111.111.111-11 requisitando vaga A1



Fonte: Autoria própria (2021)

A Figura 77 ilustra o motorista 222.222.222-22 na tentativa de requisitar a vaga A1 ao mesmo tempo que o motorista 111.111.111-11 e recebendo uma mensagem de erro. Nessa figura, é possível visualizar que a vaga A1 já estava em negociação e foi atualizado na tela do motorista 22.222.222-22 por meio das assinaturas.

Figura 77 – Motorista 222.222.222-22 requisitando vaga A1



Fonte: Autoria própria (2021)

A Figura 78 ilustra o estado do banco de dados após as requisições feitas através do sistema de gerenciamento do estacionamento. A vaga A1 que antes estava com o status *FREE*, agora está com o status *IN_NEGOTIATION*.

Figura 78 – Estado do banco de dados após requisições

spot	date	driverId	endTime	price	startTime	status
A1	2021-11-05	111.111.111-11	15:48	10	14:48	IN_NEGOTIATION
A10	2021-11-03	101.101.101-10	15:30	10	14:30	BUSY
A2	2021-11-03	121.212.121-21	15:27	10	14:27	BUSY
A3	2021-11-03	333.333.333-33	15:28	10	14:28	BUSY
A4	2021-11-03	444.444.444-44	15:28	10	14:28	BUSY
A5	2021-11-03	555.555.555-55	15:29	10	14:29	BUSY
A6	2021-11-03	666.666.666-66	15:25	10	14:25	BUSY
A7	2021-11-03	777.777.777-77	15:28	10	14:28	BUSY
A8	2021-11-03	888.888.888-88	15:34	10	14:34	BUSY
A9	2021-11-03	999.999.999-99	15:31	10	14:31	BUSY

Fonte: Autoria própria (2021)

Após essa etapa, a vaga será negociada e o processo de comunicação entre os componentes físicos descritos no experimento anterior irá acontecer.

Com base neste experimento, é possível notar que a integração em tempo real entre todos os componentes faz com que a taxa de erros de alocação de vagas diminua. Dois motoristas tentaram alocar a mesma vaga ao mesmo tempo e o sistema foi capaz de rapidamente bloquear uma das requisições e informar ao usuário.

5.5 Experimento 3

Conforme mencionado, o objeto de estudo desta seção é a variação do número de agentes vagas utilizando um Raspberry. Para tanto, foram executados 8 cenários de testes, variando o número de agentes vagas instanciados. As configurações de cada cenário de teste podem ser visualizadas na Tabela 1.

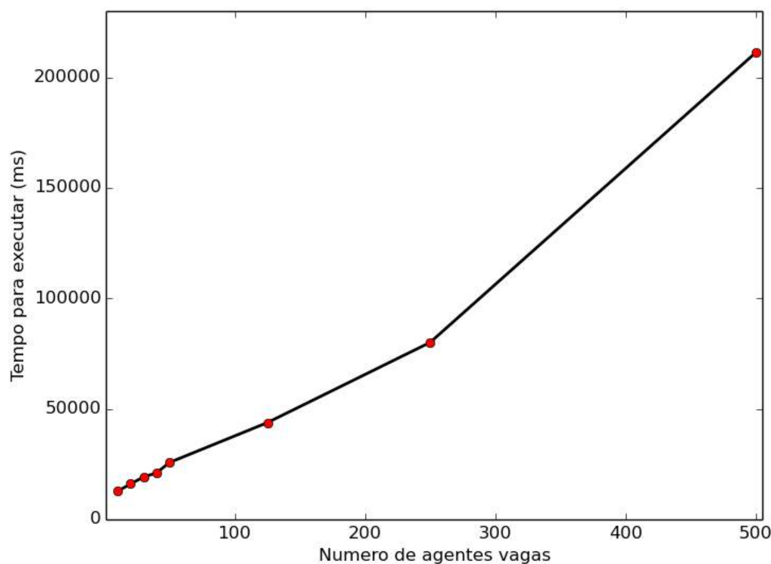
Tabela 1 – Configuração dos cenários do Experimento 3

Cenário	Número de vagas
Cenário 1	10
Cenário 2	20
Cenário 3	30
Cenário 4	40
Cenário 5	50
Cenário 6	125
Cenário 7	250
Cenário 8	500

Fonte: Autoria própria (2021)

A gráfico na Figura 79 mostra o tempo que o Raspberry levou para criar um agente gerente e todas as instâncias dos agentes vagas. A seguir, na Tabela 2 é possível visualizar os valores obtidos. No gráfico e na tabela, os valores do tempo foram medidos em milissegundos para que a análise fique mais clara e podem ser visualizados através da indicação do ponto em vermelho.

Figura 79 – Agentes criados no Raspberry



Fonte: Autoria própria (2021)

Tabela 2 – Resultados dos cenários do Experimento 3

Cenário	Número de vagas	Tempo para executar (ms)
Cenário 1	10	12730
Cenário 2	20	16120
Cenário 3	30	19190
Cenário 4	40	21140
Cenário 5	50	25800
Cenário 6	125	43800
Cenário 7	250	80106
Cenário 8	500	211224

Fonte: Autoria própria (2021)

A partir da análise dos dados coletados da execução dos cenários de testes, foi possível constatar que o tempo para instanciar as vagas do estacionamento utilizando um Raspberry tende a crescer conforme o número de vagas alocadas, como é possível verificar na Figura 79. Também foi identificado uma falha de *timeout* no comportamento do agente gerente que identifica as vagas do sistema por volta da alocação da vaga número 372, no cenário com 500 vagas.

5.6 Considerações finais

Este Capítulo apresentou os cenários de testes utilizados para analisar os resultados do desenvolvimento de um protótipo ciber-físico baseados em agentes. À partir desses resultados pode-se desenvolver o Capítulo 6, o qual apresenta a conclusão deste trabalho.

6 CONCLUSÃO

O desenvolvimento de projetos relacionados com o tema de Cidades Inteligentes são importantes, melhorando a qualidade de vida das pessoas e contribuindo no desenvolvimento sustentável das cidades. Estacionamentos Inteligentes podem contribuir neste aspecto, buscando alternativas para melhorar a qualidade de vida dos motoristas e o meio ambiente, automatizando o gerenciamento e organização de vagas de estacionamento.

Com o objetivo de criar um cenário de um estacionamento inteligente e avaliar a aplicabilidade do mesmo, foi desenvolvido um protótipo ciber-físico que detecta o estado atual da vaga e com base nessa informação, atualiza uma base de dados do estacionamento. Também foi desenvolvido um sistema de gerenciamento para permitir que o motorista realize a requisição de uma vaga para estacionar e que o administrador consiga ter controle das vagas em seu estacionamento. Foram executados cenários de testes a fim de avaliar a aplicabilidade do sistema ciber-físico e do sistema de gerenciamento do estacionamento em um ambiente que simula um estacionamento inteligente.

Com base nos resultados obtidos, foi possível analisar que o protocolo MQTT apresentou resultados satisfatórios na comunicação entre os dispositivos físicos e além disso, o sensor ultrassônico foi capaz de detectar um veículo em todos os experimentos, fazendo com que o ESP-12e publicasse corretamente a mensagem no tópico do *broker*.

O SMA desenvolvido em JADE, presente no Raspberry, também mostrou resultados satisfatórios a lidar com o gerenciamento das vagas. O agente gerente recebeu as mensagens do *broker* e replicou corretamente para o agente vaga responsável, que por sua vez atualizou o status da vaga no banco de dados e fez com que o *led* ligasse. Em um dos cenários de teste, um Raspberry não teve capacidade de lidar com 500 vagas, porém, essa arquitetura pode ser dividida entre vários Raspberrys espalhados por um estacionamento, fazendo com que apenas um Raspberry controle um número menor de vagas.

A utilização da computação em nuvem também se mostrou muito eficiente em lidar com as requisições em tempo real, não permitindo que dois motoristas requisitassem a mesma vaga ao mesmo tempo. Ainda, a arquitetura hospedada na AWS foi desenvolvida a custo zero, visto que a AWS possui um ambiente gratuito com base no

número de requisições.

Além disso, com o módulo de computação em nuvem desenvolvido, é possível realizar a integração deste módulo a um mecanismo de negociação de vagas. Com o devido ajuste ao código da negociação, é possível que o SMA, que implementa protocolos de negociação, consuma as vagas na nuvem em tempo real de forma a negociar as vagas disponíveis com os motoristas, tornando-se assim um estacionamento inteligente de fato.

6.1 Trabalhos Futuros

Este projeto de implementação de um sistema ciber-físico e um módulo utilizando computação em nuvem para conectar o motorista ao estacionamento abre caminho para outros trabalhos utilizando outras tecnologias e diferentes componentes. Destacam-se as seguintes propostas:

- Adição de novos módulos físicos ao sistema ciber-físico, de forma a testar a escalabilidade e o tempo de resposta da arquitetura;
- Desenvolvimento da arquitetura ciber-física utilizando componentes diferentes, por exemplo, substituição do Raspberry Pi ou do ESP-12e por um Arduino. Assim como a substituição do protocolo de comunicação entre os componentes, por exemplo, substituição do MQTT pelo Javino;
- Desenvolvimento do módulo em computação em nuvem utilizando outro serviço em nuvem, por exemplo a Google Cloud, de forma a comparar a usabilidade e o custo de manutenção do sistema;
- Desenvolvimento de módulos separados para atender o motorista e o administrador do estacionamento, por exemplo, um aplicativo móvel para o motorista utilizar e uma interface web para controlar o estacionamento;
- Integração do sistema de gerenciamento desenvolvido com um mecanismo de negociação de vagas (CASTRO *et al.*, 2017) (DUCHEIKO *et al.*, 2018) (ALVES *et al.*, 2019).

REFERÊNCIAS

ALVES, B. R. *et al.* Experimentation of Negotiation Protocols for Consensus Problems in Smart Parking Systems. en. In: MAŘÍK, V. *et al.* (Ed.). **Industrial Applications of Holonic and Multi-Agent Systems**. Cham: Springer International Publishing, 2019. (Lecture Notes in Computer Science), p. 189–202. ISBN 978-3-030-27878-6. DOI: 10.1007/978-3-030-27878-6_15.

ARDUINING. **Physical Computing Mini-Projects**. 2019. Disponível em: https://arduining.com. Acesso em: 5 jul. 2019.

ARTUSSE, F. **The Importance of Smart Cities**. 2019. Disponível em: https://medium.com/zify/the-importance-of-smart-cities-2a4f7f89a6cd. Acesso em: 1 jul. 2019.

AWS. **AppSync**. 2021. Disponível em: https://aws.amazon.com/pt/appsync/. Acesso em: 23 out. 2021.

AWS. **DynamoDB**. [*S. l.*], 2021. Disponível em: https://aws.amazon.com/pt/dynamodb/. Acesso em: 23 out. 2021.

AWS. **Free Tier**. 2021. Disponível em: https://aws.amazon.com/pt/free/. Acesso em: 23 out. 2021.

AWS. **Lambda**. 2021. Disponível em: https://aws.amazon.com/pt/lambda/. Acesso em: 23 out. 2021.

AWS. **Visão geral da Amazon Web Services**. 2019. Disponível em: https://docs.aws.amazon.com/pt_br/whitepapers/latest/aws-overview/introduction.html. Acesso em: 5 jul. 2019.

AWS. **What is Cloud Computing**. 2021. Disponível em: https://aws.amazon.com/what-is-%20cloud-computing. Acesso em: 23 out. 2021.

AZZOLA, F. **MQTT Protocol Tutorial: Technical description with practical Mosquitto example**. [*S. l.*], 2019. Disponível em: https://www.survivingwithandroid.com/mqtt-protocol-tutorial/. Acesso em: 4 jul. 2019.

BARRIGA, J. *et al.* Smart Parking: A Literature Review from the Technological Perspective. **Applied Sciences**, v. 9, p. 4569, out. 2019. DOI: 10.3390/app9214569.

BATTY, M. *et al.* Smart cities of the future. **The European Physical Journal Special Topics**, v. 214, p. 481–518, nov. 2012. DOI: 10.1140/epjst/e2012-01703-3.

BELLIFEMINE, F. *et al.* Developing Multi-Agent Systems with a FIPA-compliant Agent Framework. **Softw., Pract. Exper.**, v. 31, p. 103–128, fev. 2001. DOI: 10.1002/1097-024X(200102)31:2<103::AID-SPE358>3.0.CO;2-O.

BORDINI, R. H. *et al.* **Programming Multi-Agent Systems in AgentSpeak Using Jason (Wiley Series in Agent Technology)**. [*S. l.*]: John Wiley & Sons, 2007. ISBN 0470029005.

BOTELHO, P. *et al.* Proposta de implantação de um sistema ciber-físico para um Smart Parking baseado em agentes inteligentes. In: WORKSHOP-SCHOOL on Agents, Environments, and Applications. [*S. l.: s. n.*], 2019. P. 259–264.

BOTELHO, P. W. *et al.* Desenvolvimento de Aplicação para um Estacionamento Inteligente via Computação em Nuvem baseada em Agentes e Sistema Ciber-Físico. In: PROCEEDINGS of the 14th Workshop-School on Agents, Environments, and Applications (WESAAC 2020). [*S. l.*]: UTFPR, jul. 2020. v. 1, p. 185–194. ISBN 2177-2096. DOI: 10.5281/zenodo.4037413. Disponível em: https://doi.org/10.5281/zenodo.4037413.

BUTTIGIEG, S. *et al.* **Learning Node.js for Mobile Application Development**. [*S. l.: s. n.*], out. 2015. ISBN 978-1785280498.

CASTRO, L. F. S. D. *et al.* Using trust degree for agents in order to assign spots in a Smart Parking. en. **ADCAIJ: Advances in Distributed Computing and Artificial Intelligence Journal**, v. 6, n. 2, p. 45–55, jun. 2017. ISSN 2255-2863. DOI: 10.14201/ADCAIJ207624555. Disponível em: http://revistas.usal.es/index.php/2255-2863/article/view/ADCAIJ2017624555. Acesso em: 18 jul. 2017.

CHINRUNGRUENG, J. *et al.* Smart Parking: An Application of Optical Wireless Sensor Network. In: p. 66. DOI: 10.1109/SAINT-W.2007.98.

CLEMENTS, P. *et al.* Documenting Software Architectures: Views and Beyond, jun. 2002.

CODE, V. S. **Visual Studio Code**. 2021. Disponível em: https://code.visualstudio.com. Acesso em: 23 out. 2021.

DI NAPOLI, C. *et al.* Agent negotiation for different needs in smart parking allocation. In: SPRINGER. INTERNATIONAL Conference on Practical Applications of Agents and Multi-Agent Systems. [*S. l.: s. n.*], 2014. P. 98–109.

DI NAPOLI, C. *et al.* Negotiating parking spaces in smart cities. In: PROCEEDING of the 8th International Workshop on Agents in Traffic and Transportation, in conjunction with AAMAS. [*S. l.: s. n.*], 2014.

DIAS, J. *et al.* Deployment of industrial agents in heterogeneous automation environments. In: p. 1330–1335. DOI: 10.1109/INDIN.2015.7281928.

DUCHEIKO, F. F. *et al.* Implantação de um Modelo de Raciocínio e Protocolo de Negociação para um Estacionamento Inteligente com Mecanismo de Negociação Descentralizado. In: REVISTA Junior – ICCEE. [*S. l.: s. n.*], 2018. P. 25–32.

AL-FEDAGHI, S. Developing Web Applications. **International Journal of Software Engineering and Its Applications**, v. 5, mai. 2011. DOI: 10.1007/978-1-4302-3531-6_12.

FIPA. **The Foundation for Intelligent Physical Agents**. 2021. Disponível em: http://www.fipa.org. Acesso em: 5 jul. 2019.

GENG, Y. *et al.* New “Smart Parking” System Based on Resource Allocation and Reservations. **Proceedings of the IEEE Transactions on Intelligent Transportation Systems**, v. 14, n. 3, p. 1129–1139, set. 2013.

GRAPHQL. **A query language for your API**. 2021. Disponível em: https://graphql.org. Acesso em: 25 out. 2021.

GREENWOOD, D. *et al.* **Developing Multi-Agent Systems with JADE**. [*S. l.*]: John Wiley & Sons, 2004.

GROKHOTKOV, I. **ESP8266 Arduino Core Documentation**. 2019. Disponível em: https://readthedocs.org/projects/arduino-esp8266/downloads/pdf/latest/. Acesso em: 4 jul. 2019.

HARRINGTON, W. **Learning Raspbian**. [*S. l.: s. n.*], fev. 2015.

HASSOUNE, K. *et al.* Smart parking systems: A survey. In: IEEE. 2016 11th International Conference on Intelligent Systems: Theories and Applications (SITA). [*S. l.: s. n.*], 2016. P. 1–6.

JADE. **Jade**. 2003. Disponível em: https://jade.tilab.com/. Acesso em: 5 jul. 2019.

JADHAV, M. A. *et al.* Single Page Application using AngularJS. **International Journal of Computer Science and Information Technologies**, v. 6, n. 3, p. 2876–2879, 2015.

JAZDI, N. Cyber physical systems in the context of Industry 4.0. **Proceedings of the IEEE International Conference on Automation, Quality and Testing, Robotics (AQTR’14)**, p. 1–4, mai. 2014.

KAITHAN, S. K. *et al.* Design Techniques and Applications of Cyberphysical Systems: A Survey. **IEEE Systems Journal**, p. 350–365, jun. 2015.

KHANNA, A. *et al.* IoT based smart parking system. **International Conference on Internet of Things and Applications (IOTA)**, p. 266–270, jan. 2016.

LAZARIN, N. M. *et al.* A Robotic-agent Platform for Embedding Software Agents Using Raspberry Pi and Arduino Boards. In: 9th Software Agents, Environments and Applications School. [*S. l.: s. n.*], 2015.

LIN, T. *et al.* A survey of smart parking solutions. **IEEE Transactions on Intelligent Transportation Systems**, IEEE, v. 18, n. 12, p. 3229–3253, 2017.

LIU, D. *et al.* How Much Can a Smart Parking System Save You? In: ERGONOMICS in Design The Quarterly of Human Factors Applications. [*S. l.: s. n.*], 2014. P. 15–20.

MADSEN, M. *et al.* Static Analysis of Event-Driven Node.js JavaScript Applications. **ACM SIGPLAN Notices**, v. 50, p. 505–519, out. 2015. DOI: 10.1145/2858965.2814272.

MCLAREN, D. *et al.* Smart for a reason. In: [*s. l.: s. n.*], out. 2018. P. 169–181. ISBN 9781351182409. DOI: 10.4324/9781351182409-13.

MELL, P. *et al.* The NIST Definition of Cloud Computing. **National Institute of Standards and Technology**, NIST, 2011.

MELLADO, A. *et al.* Um Módulo de Precificação Dinâmica em Sistema Multiagente de um Estacionamento Inteligente. pt. **Revista Eletrônica de Iniciação Científica em Computação**, v. 19, n. 1, set. 2021. Number: 1. ISSN 1519-8219. Disponível em: https://sol.sbc.org.br/journals/index.php/reic/article/view/1779. Acesso em: 24 set. 2021.

MIMBELA, L. E. Y. *et al.* **A Summary of Vehicle Detection and Surveillance Technologies used in Intelligent Transportation Systems**. [*S. l.*]: Southwest Technology Development Institute (SWTDI) at New Mexico State University (NMSU), 2000.

MQTT. **The Standard for IoT Messaging**. 2019. Disponível em: https://mqtt.org. Acesso em: 5 jul. 2019.

PANTOJA, C. E. *et al.* ARGO: An Extended Jason Architecture that Facilitates Embedded Robotic Agents Programming. en. In: BALDONI, M. *et al.* (Ed.). **Engineering Multi-Agent Systems**. Cham: Springer International Publishing, 2016. (Lecture Notes in Computer Science), p. 136–155. ISBN 978-3-319-50983-9. DOI: 10.1007/978-3-319-50983-9_8.

RASPBERRY. **The Raspberry Pi**. 2019. Disponível em: https://www.raspberrypi.com. Acesso em: 5 jul. 2019.

REACT. **ReactJS**. 2021. Disponível em: https://reactjs.org. Acesso em: 23 out. 2021.

RIZVI, S. R. *et al.* ASPIRE: An Agent-Oriented Smart Parking Recommendation System for Smart Cities. **IEEE Intelligent Transportation Systems Magazine**, IEEE, 2018.

RUSSELL, S. *et al.* Artificial Intelligence: A Modern Approach. CUMINCAD, 1995.

SAKURADA, L. *et al.* Development of Agent-Based CPS for Smart Parking Systems. In: IECON 2019 - 45th Annual Conference of the IEEE Industrial Electronics Society. [*S. l.: s. n.*], out. 2019. v. 1, p. 2964–2969. ISSN: 1553-572X. DOI: 10.1109/IECON.2019.8926653.

SANISLAV, T. Cyber-Physical Systems - Concept, Challenges and Research Areas. **Control Engineering and Applied Informatics**, v. 14, p. 28–33, mai. 2012.

SHAH, K. *et al.* Arduino Based Smart Parking System. In: INTERNATIONAL Research Journal of Engineering and Technology. [*S. l.: s. n.*], 2017. P. 882–884.

SJOGELID, S. **Raspberry Pi for Secret Agents**. [*S. l.: s. n.*], abr. 2013. ISBN 978-1849695787.

ŠKRABA, A. *et al.* Prototype of group heart rate monitoring with NODEMCU ESP8266. In: p. 1–4. DOI: 10.1109/MECO.2017.7977151.

SYSTEMS, E. **Espressif**. 2019. Disponível em: https://www.espressif.com/en/products/hardware. Acesso em: 5 jul. 2019.

TECHNOLOGIES, C. **Product User's Manual – HC-SR04 Ultrasonic Sensor**. 2019. Disponível em: http://web.eece.maine.edu/~zhu/book/lab/HC-SR04%5C%20User%5C%20Manual.pdf. Acesso em: 4 jul. 2019.

TECHOPEDIA. **Web Based Application**. 2021. Disponível em: https://www.techopedia.com/definition/26002/web-based-application. Acesso em: 23 out. 2021.

THANGAM, E. C. *et al.* Internet of Things (IoT) based Smart Parking Reservation System using Raspberry-pi. In.

UPTON, E. *et al.* **Raspberry Pi User Guide**. [*S. l.: s. n.*], dez. 2013. ISBN 978-1118795484.

WOOLDRIDGE, M. **An introduction to multiagent systems**. [*S. l.*]: John Wiley & Sons, 2009.