UNIVERSIDADE TECNOLÓGICA FEDERAL DO PARANÁ

DEPARTAMENTO DE INFORMÁTICA

BACHARELADO EM CIÊNCIA DA COMPUTAÇÃO

RAFAEL DA SILVA KOTESKI

IMPLEMENTAÇÃO DE UM SISTEMA MULTIAGENTE PARA SIMULAÇÃO DA CONDUÇÃO DE TRENS DE CARGA USANDO O *FRAMEWORK* JADE

TRABALHO DE CONCLUSÃO DE CURSO

PONTA GROSSA

2019

RAFAEL DA SILVA KOTESKI

IMPLEMENTAÇÃO DE UM SISTEMA MULTIAGENTE PARA SIMULAÇÃO DA CONDUÇÃO DE TRENS DE CARGA USANDO O FRAMEWORK JADE

Trabalho de Conclusão de Curso apresentado como requisito parcial à obtenção do título de Bacharel em Bacharelado em Ciência da Computação, do Departamento de Informática, da Universidade Tecnológica Federal do Paraná.

Orientador: Prof. Dr. André Pinz Borges

PONTA GROSSA

2019

Ministério da Educação
Universidade Tecnológica Federal do Paraná
Câmpus Ponta Grossa
Diretoria de Graduação e Educação Profissional
Departamento Acadêmico de Informática
Bacharelado em Ciência da Computação

TERMO DE APROVAÇÃO

IMPLEMENTAÇÃO DE UM SISTEMA MULTIAGENTE PARA SIMULAÇÃO DA CONDUÇÃO DE TRENS DE CARGA USANDO O FRAMEWORK JADE

por

RAFAEL DA SILVA KOTESKI

Este Trabalho de Conclusão de Curso (TCC) foi apresentado em 13 de novembro de 2019 como requisito parcial para a obtenção do título de Bacharel em Ciência da Computação. O candidato foi arguido pela Banca Examinadora composta pelos professores abaixo assinados. Após deliberação, a Banca Examinadora considerou o trabalho aprovado.

_____
Prof. Dr. André Pinz Borges
Orientador

_____
Prof. Dr. André Koscianski
Membro titular

_____
Prof. Dr. Gleifer Vaz Alves
Membro titular

_____
Prof. MSc. Geraldo Ranthum
Responsável pelo Trabalho de Conclusão de Curso

_____
Prof(a). Dra. Mauren Louise Sguario
Coordenador do curso

- A Folha de Aprovação assinada encontra-se arquivada na Secretaria Acadêmica -

AGRADECIMENTOS

À Deus, por todas as bênçãos até aqui concedidas.

Ao professor André Pinz Borges, por toda orientação necessária e conhecimento adquirido durante a realização deste trabalho, sendo o principal colaborador para sua realização. Ao professor André, o meu muito obrigado.

Aos amigos adquiridos durante o período de faculdade, por todos os momentos de diversão e horas de estudos juntos. Amizade verdadeira que levarei para toda a vida.

Por fim agradeço à minha família, amigos e minha amada namorada, pela compreensão de não poder estar presente em muitos momentos e a todo o apoio dado durante a graduação.

RESUMO

KOTESKI, Rafael da Silva. **Implementação de um sistema multiagente para simulação da condução de trens de carga usando o framework JADE.** 2019. 102 f. Trabalho de Conclusão de Curso (Bacharelado em Ciência da Computação) - Universidade Tecnológica Federal do Paraná. Ponta Grossa, 2019.

A simulação de viagens no modal férreo caracteriza-se por ser uma tarefa que demanda diversos recursos para ser realizada, como utilização de locomotivas, vagões, uso de uma via férrea e alocação de pessoas para controlar o processo. Pensando nisto, este trabalho tem como objetivo implementar um Sistema MultiAgente (SMA) que possa ser utilizado na simulação de condução de trens de carga utilizando o *framework* JADE. Cada agente do SMA é dotado da capacidade de simular a condução de trens de carga, realizando os principais cálculos e procedimentos necessários para simular as ações aplicadas por um maquinista durante a condução. Foram gerados seis cenários, sendo cada um constituído de composições de diferentes configurações. Esses cenários foram analisados em termos de quantidade de mensagens trocadas, fluxo de mensagens e tempo total de viagem.

Palavras-chave: Condução de trens. Agente. Framework. JADE

ABSTRACT

KOTESKI, Rafael da Silva. **Implementation of a MultiAgent System for freigth train driving simulation using JADE framework.** 2019. 102 s. Work of Conclusion Course (Graduation in Computer Science) - Federal Technology University - Paraná. Ponta Grossa, 2019.

Rail-mode simulation travel is a task that requires several resources to be performed, such as the use of railcars, railroad and the allocation of people to control the process. With this in mind, this work aims to implement a MultiAgent System (MAS) that can be used to simulate freight train driving using JADE framework. Each MAS agent has the ability to simulate the driving of freight trains, performing the main calculations and necessary procedures to simulate the actions taken by a drive while driving. Six scenarios were generated, each consisting of compositions of different configurations. These scenarios were analyzed in terms of number of messages exchanged, messages flow and total travel time.

Keywords: Driving trains. Agent. Framework. JADE

LISTA DE ILUSTRAÇÕES

LISTA DE CÓDIGOS

LISTA DE TABELAS

LISTA DE GRÁFICOS

LISTA DE ABREVIATURAS E SIGLAS

ACC	*Agent Communication Channel*
ACL	*Agent Communication Language*
AMS	*Agent Management System*
AP	*Agent Platform*
BDI	*Belief*, *Desire*, *Intention*
CT	*Container Table*
DF	*Directory Facilitator*
FIPA	*Foundation for Intelligence Physical Agents*
GADT	*Global Agent Descriptor Table*
HP	*Horse Power*
IA	Inteligência Artificial
JADE	Java *Agent DEvelopment Framework*
POO	Programação Orientada a Objetos
RBC	Raciocínio Baseado em Casos
SMA	Sistema Multiagentes

SUMÁRIO

1 INTRODUÇÃO

A condução de uma locomotiva é uma tarefa difícil que exige do maquinista conhecimento, habilidade e experiência para uma condução segura e eficiente. Uma condução é dita segura quando não causa danos à via ou ao trem. A eficiência dá-se quando o consumo de combustível é o menor possível em consequência de ações corretas. O modo de condução de um trem tem direta influência em seu consumo de combustível, visto que a aplicação de um ponto de aceleração (elemento simular à uma marcha de um veículo de passeio) gera uma determinada potência necessária para movimentar o trem e, consequentemente, um consumo proporcional.

A troca de pontos de aceleração não é obrigatoriamente sequencial como em um veículo de passeio, mas nem por isso a condução torna-se um processo fácil. Há uma dificuldade em se definir um padrão de condução da locomotiva, pois não há, por exemplo, um ponto de aceleração específico que sempre possa ser empregado em um determinado ponto da via de forma eficiente (BORGES, 2015). Logo, a escolha do ponto de aceleração que resulte no menor consumo possível é uma tarefa que exige experiência do maquinista, esta adquirida à medida que novas e diferentes viagens são realizadas.

A problemática de condução de trens de carga pode ser solucionada por meio de técnicas computacionais com a capacidade de tomar decisões baseadas em objetivos a serem alcançados, como Sistemas Multiagentes. Um Sistema MultiAgente (SMA) pode ser definido como um sistema formado por agentes, *software* que detém a capacidade de captar informações do meio ao qual está inserido e reagir a elas de forma independente, baseando suas ações a partir de objetivos finais impostos a ele (WOOLDRIDGE, 2009). Baseando-se na ideia de que a inteligência do sistema surge do comportamento social dos componentes que integram o ambiente, estes sistemas podem servir de modelo para sistemas reais complexos e adotar a possibilidade de os agentes colaborarem uns com os outros visando atingir metas globais (COSTA, 1997).

Os SMAs podem ser desenvolvidos com a utilização de *frameworks* que auxiliem na sua implementação, como o *framework* JADE (BELLIFEMINE; CAIRE; GREENWOOD, 2007). O *Framework* JADE, JAVA Agent DEvelopment Framework, é um *software* desenvolvido em linguagem JAVA com o objetivo de simplificar a

implementação de SMA por meio de um *middleware*, o qual traz um conjunto de funções e ferramentas gráficas que dão suporte ao processo de implementação de agentes. Sendo um programa *open source*, JADE oferece um simples e poderoso modelo de composição e execução de tarefas simples, além de comunicação ponto a ponto baseada em envio de mensagens assíncronas e de outros recursos que possibilitam o desenvolvimento de SMAs (TELECOM ITALIA, 2019).

Neste trabalho foi realizada a implementação de um SMA com a habilidade de condução de trens de carga. A partir do *framework* JADE, um agente controlador e um agente condutor foram desenvolvidos. O agente controlador é responsável por manter uma lista atualizada das posições de cada agente condutor presente na via e liberar viagem aos condutores. O agente condutor realiza a condução do trem de carga, a partir de cálculos definidos para simulação das condições de condução. O SMA é composto por um agente controlador e *n* agentes condutores, com os condutores realizando sua condução em uma mesma via e o controlador mantendo um controle sobre os agentes condutores existentes.

Testes foram realizados a partir de cenários, sendo cada cenário composto por um agente controlador e um determinado número de agentes condutores, com cada um deles apresentando diferentes configurações de trem, como peso e quantidade de veículos. Os resultados gerados foram analisados em termos de quantidade de mensagens trocadas entre agente condutor e agentes controladores.

1.1 OBJETIVO

O objetivo geral deste trabalho é implementar um Sistema MultiAgente condutor de trens de carga usando o *framework* JADE, a fim de simular o processo de condução de trens de carga.

1.2 OBJETIVOS ESPECÍFICOS

Definiu-se os seguintes objetivos específicos para melhor se alcançar o objetivo geral:

- Analisar o domínio do problema acerca da condução de um trem de carga, como conhecimentos e habilidades necessárias para uma boa condução;
- Implementar um agente inteligente em JADE com cálculos e procedimentos necessários para a condução;
- Adaptar o agente desenvolvido para funcionamento em um SMA;
- Analisar os resultados em termos de quantidade de mensagens trocadas pelos agentes e fluxo das mensagens.

1.3 JUSTIFICATIVA

O modal férreo tem sua suma importância no país desde os anos 1854, quando começou a ser implantada no país, com a construção da Estrada de Ferro Mauá. O transporte ferroviário se destaca pela elevada capacidade de carga possível a ser transportada, meio de transporte mais adequado para médias e longas distâncias, baixo custo de transporte, baixo custo de manutenção, além de possuir mais segurança em relação ao modal rodoviário, visto o baixo índice de acidentes, furtos e roubos (PEGÔ FILHO, 2016).

O modal férreo, por ser viável para o transporte de carga, ainda possui espaço para redução de gastos, o que incentiva pesquisas e inovação neste setor. Assim sendo, a utilização de qualquer recurso tecnológico capaz de reduzir, por exemplo, a quantidade de combustível consumido, pode auxiliar na diminuição significativa dos custos anuais de operação (BORGES, 2015).

Alguns problemas tornam-se evidentes na condução de trens de carga, como o alto consumo de combustível e, consequentemente, maior emissão de poluentes na atmosfera devido à queima de combustíveis fósseis (DORDAL et al, 2011). A complexidade da condução de uma locomotiva é alta para maquinistas inexperientes, porém diminui à medida em que estes adquirem experiências e realizando novas viagens. Pequenas variações nos objetos que compõem este ambiente fazem com que as tomadas de decisões variem de forma significativa, tornando o modal férreo um ambiente amplamente dinâmico. Para melhorar a quantidade de combustível consumida durante um percurso, métodos computacionais são estudados para

identificar a melhor decisão a ser tomada durante a condução de um trem, como o simulador para condução assistida implementado por Borges (2015).

Com a utilização do simulador para a condução de trens de carga, torna-se evidente as múltiplas vantagens proporcionadas. Novas técnicas de condução poderão ser implementadas, analisadas e aprimoradas, proporcionando vantagens econômicas pela diminuição de combustível gasto durante uma viagem, além das vantagens ambientais, visto a diminuição de emissão de gases poluentes na atmosfera. Há também vantagens cientificas, principalmente para a área da Ciência da Computação, como um ambiente que permitirá a implementação e testes de algoritmos em um problema com características reais.

Em SMA não existe um controle centralizado para a resolução do problema. Assim sendo, as tomadas de decisões são realizadas e divididas pelos próprios agentes, garantindo ao sistema a capacidade de adaptar-se mais facilmente a novas situações como a inclusão ou exclusão tanto de novos agentes, quanto de mudanças que possam vir a acontecer no ambiente aos quais estes estão inseridos. Esse tipo de organização permite a implementação de agentes de modo independente a problemas específicos que possam ocorrer, sendo definidos protocolos que possam ser utilizados em situações genéricas que ocorram ao ambiente (HÜBNER, 2003).

O uso de agentes para condução de trens de carga se dá principalmente pela troca de mensagens entre os mesmos e pela habilidade de autonomia, não sendo necessária intervenção humana em sua execução. Para um ambiente onde há mais de um trem para locomoção, é indispensável a comunicação entre ambos ou então a troca de informações por parte dos trens com um componente principal, para que haja o controle sobre os trens e sobre a via, evitando assim cenários indesejados como colisão, por exemplo.

1.4 ESCOPO

Este trabalho é continuação das pesquisas realizadas por Borges (2015), onde o foco de estudo foi a geração de planos de ações para rebocar trens de carga em vias férreas reais utilizando um agente capaz de auxiliar na condução de trens. Aqui, o objetivo é implementar em JADE um Sistema Multiagente para simulação de

um modal férreo. No sistema cada agente é dotado da capacidade de conduzir trens de carga com base apenas em cálculos matemáticos obtidos da literatura. Não será o foco aqui: (i) o desenvolvimento e estudo aprofundado de técnicas de condução de trens; (ii) o desenvolvimento de técnicas capazes de otimizar os resultados já conhecidos na literatura; (iii) o emprego de seções de bloqueio e semáforos para garantir a distância mínima de segurança entre duas composições; e (iv) a interação com um computador de bordo para leitura e aplicação das ações dos agentes.

Trabalhos são desenvolvidos visando a otimização na condução de trens de carga, como os trabalhos desenvolvidos por (STREISKY, 2019) e (BERNARDO, 2019). Nestes, técnicas de IA são aplicadas para aprimorar as decisões tomadas pelo agente condutor no momento de sua condução, resultando em um menor consumo de combustível e tempo de viagem se comparado a condução de um maquinista.

1.5 ORGANIZAÇÃO

Este trabalho está dividido em 5 capítulos. No Capítulo 2 é apresentado o referencial teórico, abordando temas sobre agentes e SMAs, condução assistida de trens de carga e sobre *framework* JADE. No Capítulo 3, é abordado os passos do desenvolvimento do trabalho proposto. No Capítulo 4 é apresentado os resultados gerados a partir do sistema desenvolvido. No Capítulo 5, é descrito as conclusões finais.

2 REFERENCIAL TEÓRICO

Neste capítulo será abordado o embasamento teórico utilizado para a realização deste trabalho. Na seção 2.1, é abordado sobre agentes e sistemas multiagentes. Na seção 2.2, é abordado sobre condução assistida com exemplo dos cálculos utilizados. Na seção 2.3 é abordado sobre o *framework* JADE e seu funcionamento. Por fim, na seção 2.4 são apresentados os trabalhos relacionados.

2.1 SISTEMAS MULTIAGENTES

Nesta seção será abordado Sistemas Multiagentes, trazendo definições sobre o que é um Sistema Multiagente e o que o compõe, principais definições sobre um agente e seu funcionamento, além das quatro principais características para que um elemento seja considerado um agente.

2.1.1 Agentes

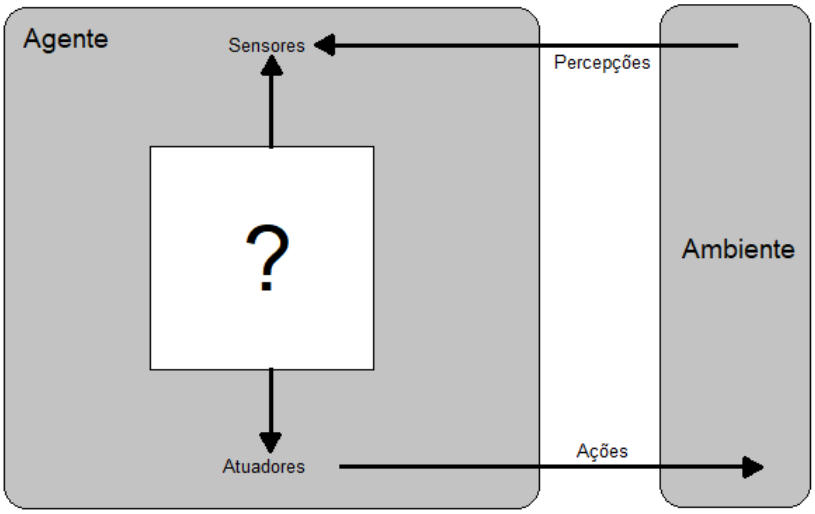
Considerado um dos mais importantes paradigmas de programação, podendo melhorar os métodos de conceitualização, projeção e implementação de *softwares* (BELLIFEMINE; CAIRE; GREENWOOD, 2007), não há uma definição exata sobre agente (COSTA, 1997).

Embora o termo agente seja utilizado de uma maneira vaga, é possível determinar uma definição comum sobre. Agente é uma entidade virtual ou real com a capacidade de atuar em um ambiente e comunicar-se diretamente com outros agentes. Agente é guiado por um objetivo individual, possuindo habilidades próprias (comportamentos) e podendo oferecer serviços a partir destes. Seus comportamentos tendem a satisfazer os objetivos a serem alcançados, tendo como base os recursos e comportamentos disponíveis, suas percepções do ambiente e mensagens recebidas (WOOLDRIDGE, 2009).

A Figura 1 representa um agente interagindo com o ambiente, obtendo informações através de seus sensores, raciocinando com base nestas informações

(operação representada pelo símbolo “?”) e interagindo com o meio através de seus atuadores.

Figura 1 - Representação de um agente interagindo com o ambiente



Fonte: adaptado de Castro (2015).

Pode se estabelecer, a partir da estrutura apresentada, que múltiplos agentes sejam utilizados para resolução de problemas de grande complexidade, onde cada agente se utiliza do paradigma mais adequado na solução de determinada particularidade do problema (CASTRO, 2015).

Um agente deve ser dotado da capacidade de comunicação com outros agentes, formando assim uma rede de troca de informações onde há cooperação para descoberta da solução do problema. O que também dita a capacidade de certo agente para solucionar problemas de forma eficiente é o conhecimento a ele atribuído, além de recursos de *hardware* disponíveis (SYCARA, 1998).

A cooperação entre agentes se dá pela capacidade de troca de informações entre eles, não sendo necessariamente cooperatividade em si, como por exemplo compartilhamento de recursos vantajosa para ambos agentes (CASTRO, 2015).

Existem características importantes que ditam se um determinado elemento computacional é um agente. São essas características que, atreladas a capacidade de comunicação, fazem com que agentes sejam utilizados para solucionar problemas onde há necessidade de interação humana no ambiente aos quais serão inseridos (WOOLDRIDGE, 2009), sendo elas:

- *Autonomia*: operam sem intervenção humana, com total controle sobre suas ações e estados internos. Um agente autônomo tem competência para decidir qual é a abordagem ideal para solucionar um problema utilizando suas percepções do meio e seu conhecimento para tomadas de decisões;
- *Habilidade social*: cooperação com outros agentes com o intuito de realizar suas tarefas. Agentes demandam trocar informações a todo momento para coleta de informações sobre o ambiente e de outros agentes. A comunicação entre agentes pode ser dividida em 3 etapas: comunicação, cooperação e negociação;
- *Reatividade*: capacidade de perceber o meio que se encontra e responder adequadamente às mudanças que ocorrem. Agentes podem ter dois tipos de reatividade, similar ao reflexo humano ou então com um certo nível de cognição. Atribuir essas duas características podem ser de alta complexidade, uma vez que o ambiente a ser trabalhado pode ser altamente dinâmico;
- *Proatividade*: aptidão em tomar iniciativas por conta própria, tomadas a partir de objetivos pré-definidos.

2.1.2 Classificação

Agentes são utilizados para resolver os mais diversos tipos de problemas, por isso cada agente possui características próprias para solucioná-los. Pesquisas iniciais na utilização de agentes concentraram-se principalmente no desenvolvimento de arquiteturas, estas variando de arquiteturas puramente reativas, que operam de forma simples em um formato estímulo-resposta, a arquiteturas mais decisivas, raciocinando sobre suas ações. Entre essas duas técnicas, estão variações híbridas que tentam envolver tanto a reação quanto a decisão em um esforço em selecionar o melhor de cada abordagem (BELLIFEMINE; CAIRE; GREENWOOD, 2007).

Os quatro tipos de agentes são (BELLIFEMINE; CAIRE; GREENWOOD, 2007): (i) reativos, (ii) deliberativos, (iii) racionais e (iv) híbridos.

2.1.2.1Agentes reativos

Agentes reativos implementam tomadas de decisões como um mapeamento direto de uma situação para sua respectiva ação, sendo estas baseadas em um mecanismo de estímulo-resposta sem conhecimento nem memória de suas ações. São agentes bem simples que baseiam suas reações a partir de suas percepções do ambiente, não sendo necessário a utilização de nenhum raciocínio complexo de símbolos (WOOLDRIDGE, 2009).

A vantagem desta abordagem se dá pela boa performance em ambientes dinâmicos, assim como sua fácil implementação em relação a outras abordagens. Sua desvantagem surge pela complexidade em desenvolver agentes para realizar tarefas específicas, principalmente quando é exigido uma grande quantidade de comportamentos (BELLIFEMINE; CAIRE; GREENWOOD, 2007).

2.1.2.2Agentes deliberativos

Agentes deliberativos possuem uma representação simbólica do meio em que estão inseridos. Suas decisões (ou deliberações) são feitas por meio de um processo baseado em raciocínio lógico em que se um agente tem conhecimento de que uma de suas ações o levará a um de seus objetivos, o mesmo executará essa ação (WOOLDRIDGE, 2009).

2.1.2.3Agentes racionais

Arquiteturas racionais ou cognitivas tomam suas decisões de ações baseadas em um conjunto de raciocínios lógicos, a partir de uma representação simbólico do ambiente (WOOLDRIDGE, 2009).

Um modelo computacional utilizado em agentes racionais é BDI. O modelo BDI (*Belief*, *Desire*, *Intention*), do inglês “Confiança, Crença, Intenção”, promove uma lógica que define atitudes mentais de crenças, desejos e intenções ao agente, sendo as ações tomadas por este baseadas no conteúdo desses estados (BELLIFEMINE; CAIRE; GREENWOOD, 2007)

Crença é as informações que o agente tem sobre o ambiente ao qual está inserido. Desejo é os possíveis estados que o agente pretende atingir, não significando que estes serão alcançados. Intenção é os objetivos atribuídos ao agente, os quais serão alcançados a partir dos desejos implementados (HEIJMEIJER, 2016).

Diferentes sistemas baseados em agentes implementaram BDI em diversas aplicações, demonstrando a viabilidade do modelo (BELLIFEMINE; CAIRE; GREENWOOD, 2007).

2.1.2.4Agentes híbridos

Arquiteturas híbridas, também conhecidas como arquiteturas em camadas, permite comportamento tanto de agente reativo quanto de agente racional. Para poder utilizar os dois métodos em um único agente, são utilizados subsistemas organizados de acordo com camadas de uma hierarquia. Existem dois tipos de fluxo de controle em agente híbrido, sendo eles camada horizontal e camada vertical.

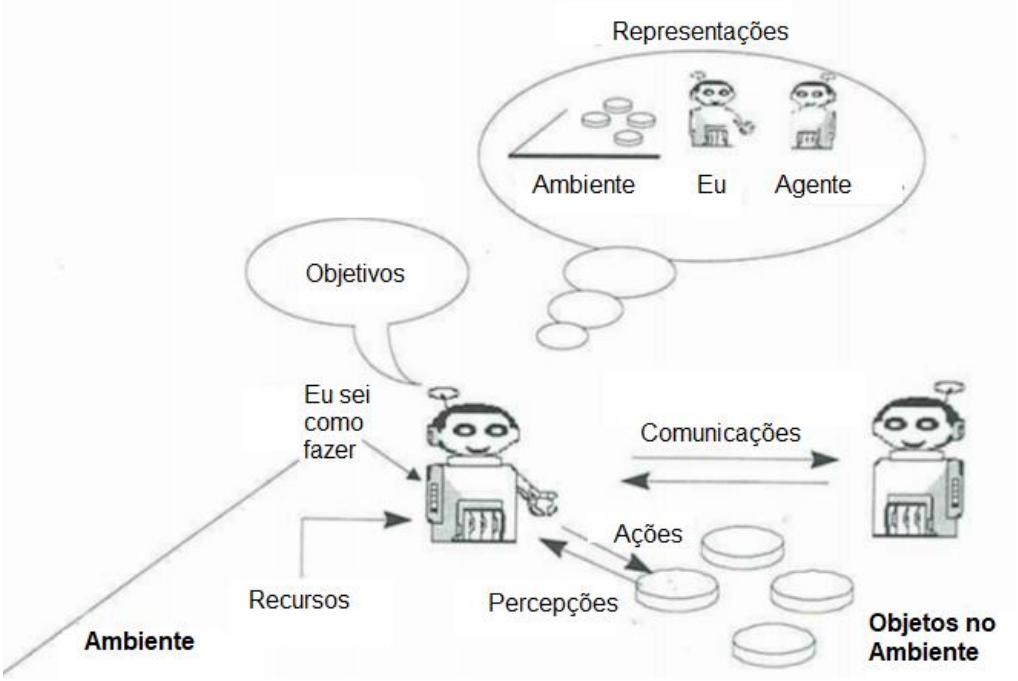
Na camada horizontal, as camadas são diretamente ligadas aos sensores e atuadores, cada camada agindo assim como um agente distinto. Camada vertical pode ser dividida em uma arquitetura de uma passagem e arquitetura de duas passagens, sendo arquitetura de uma passagem uma disposição em que o controle das ações flui da camada inicial (leitura dos dados pelos sensores) para a camada final (tomada de decisões) e, arquitetura de duas passagens uma disposição onde os dados fluem na sequência de camadas determinada e então flui novamente para a camada inicial, reduzindo assim a interação entre as camadas implementadas (BELLIFEMINE; CAIRE; GREENWOOD, 2007).

2.1.3 Sistema Multiagente

Sistema Multiagente (SMA) é o nome dado à área da Inteligência Artificial (IA) que estuda o comportamento de um conjunto de agentes em um mesmo ambiente, cujo objetivo é solucionar um dado problema que está além de suas capacidades individuais (O’HARE; JENNINGS, 1996). A comunicação dentro desse sistema é

crucial e permite que um agente obtenha informações de outros agentes presentes no ambiente (HEIJMEIJER, 2016), conforme ilustra a Figura 2.

Figura 2 - Representação de um agente interagindo com o ambiente e outros agentes



Fonte: adaptado de Ferber (1999).

Um dos focos de estudo da área de IA é o desenvolvimento de sistemas que se aproximam da realidade tomando como base as visões que outras áreas do conhecimento têm da mesma, como por exemplo o desenvolvimento de Redes Neurais, baseado no estudo da Biologia. Assim sendo, o desenvolvimento de SMA tem como base o estudo da Sociologia, relacionando uma concepção de sistema com propriedades que somente sociedades possuíam até então (FERBER, 1999).

Diferente de outros paradigmas da IA que focam seus estudos em um único indivíduo, a característica predominante de SMA é a coletividade. Em SMA, deixa-se de lado a tentativa de entender e reproduzir o comportamento humano de forma isolada, focando seus estudos para a forma de interação entre os agentes que integram o sistema e sua organização (JOHNSON, 2001).

O ciclo de um SMA pode ser dividido em duas etapas: (i) concepção e (ii) resolução. Na fase de concepção, modelos de objetivo geral, de interações e de formas de organização (métodos para solucionar o problema) são definidos aos

agentes. Na fase de resolução, um grupo de agentes executa os modelos criados na etapa anterior para resolver os problemas encontrados (WOOLDRIDGE, 2009).

Diferentes tipos de problemas exigem diferentes escolhas de modelos por parte dos agentes. Assim sendo, os modelos não são desenvolvidos para solucionar um problema específico, mas desenvolvidos de maneira que possam ser aplicáveis a vários tipos de problemas (JENNINGS; WOOLDRIDGE, 1998).

SMA são usados cada vez mais em uma variedade de aplicações, desde pequenos sistemas para assistência pessoal até complexos e críticos sistemas para aplicações industriais. Aplicações industriais são relevantes para SMA pois foi em uma aplicação como esta onde foi testada as primeiras técnicas de SMA, demonstrando o grande potencial de sua utilização. Uma das grandes áreas de aplicação de SMA é o gerenciamento de informações (JENNINGS; WOOLDRIDGE, 1998).

2.2 CONDUÇÃO ASSISTIDA

Condução assistida é definido como um programa computacional com o intuito de auxiliar as tomadas de ações por parte do maquinista ou até mesmo conduzir um trem de forma autônoma (BORGES, 2015).

A percepção da via se dá pela leitura de sensores presentes no trem. Estes sensores geram dados ao trem os quais são interpretados pelo programa condutor. Os dados podem ser adquiridos no início de uma viagem por meio da ordem de despacho – número de locomotivas e vagões, posição inicial da viagem e perfil da via – ou durante a condução, através da leitura dos sensores do trem em intervalos de tempo – velocidade, posição, tempo de viagem (BORGES, 2015).

A leitura dos dados obtidos pelos sensores não é suficiente por si só para uma correta escolha de determinada ação. Diversos cálculos devem ser realizados a fim de compor a situação atual e presumir uma futura situação do trem conduzido após a aplicação de determinada ação. Em uma condução assistida, estes cálculos são realizados com o intuito de verificar situações adversas, como excessos ou desperdícios.

O Anexo A apresenta as equações utilizadas para a condução de trens de carga. De modo geral, este procedimento se resume em encontrar e aplicar uma força

maior que a resistência total de um trem para que o deslocamento seja possível. Para movimentar um trem, é preciso que (i) a força de tração seja superior a resistência total, (ii) a força de aceleração seja positiva, (iii) a velocidade seja menor que a velocidade máxima permitida do trecho percorrido e (iv) a potência gerada pelo ponto de aceleração aplicado seja maior que a potência mínima necessária para deslocar o trem (BORGES, 2015).

2.2.1 Exemplo de Aplicação dos Cálculos

Seja *T* um trem com 1 locomotiva e 4 vagões, onde suas características são descritas nas tabelas seguintes. A Tabela 1 apresenta dados de configuração de uma locomotiva e de um vagão. A apresenta dados de potência e consumo de cada ponto de aceleração da locomotiva. A Tabela 3 apresenta dados de cada ponto de medida de uma possível via.

Tabela 1 - Dados de configuração de uma locomotiva e de um vagão

Característica	Locomotiva	Vagão
Comprimento	20m	20m
Peso	169,7t	99,46t
Número de eixos	6	4
Área frontal	120 pés	120 pés

Fonte: Borges (2015).

Tabela 2 - Potência e consumo referente a cada ponto de aceleração

Ponto de Aceleração	Potência (*HP*)	Consumo (L/min)
Neutro	0	0,3168
1	100	0,5678
2	275	1,0668
3	575	1,9500
4	460	3,0330
5	1440	4,5330
6	1930	6,1830
7	2500	7,6998
8	2940	9,4002

Fonte: Borges (2015).

Tabela 3 - Exemplos de ponto de medida

ID	KM Atual	Velocidade máxima (em km/h)
1	349	60
2	349	60
3	349	60
4	349	60
5	349	60
6	350	60
7	350	60
8	350	60
9	350	60
10	350	60
11	351	60
12	351	60
13	351	60
14	351	60

Fonte: Borges (2015).

Para exemplificação dos dados, foi considerado um trecho em linha reta de uma via férrea representada pelos dados da Tabela 3, sendo cada linha um ponto de medida com 20 metros. Inicialmente, o último vagão está posicionado no ponto de medida 1 e a locomotiva está posicionado no ponto de medida 5. O processo consiste em deslocar o trem do ponto de medida 5 até o ponto de medida 10. Ao final do processo, o trem terá um deslocamento de 100 metros. O Quadro 1 apresenta as etapas do cálculo. A Tabela 4 apresenta os resultados gerados.

Quadro 1 - Etapas dos cálculos para deslocar um trem

Ordem	Etapa			Variáveis
1	Loop *i*	Percepção		pm, v
2		Cálculo de Resistências		R_t
3		Loop *j*	Seleção de Ponto de Aceleração	pa
4			Cálculo de Forças	F_{ac}
5		Atuação		pa
6		Memória		Δ_l, Δ_t, C

Legenda:

C: Consumo

F_{ac}: Força de aceleração

pa: Ponto de aceleração

pm: Ponto de medida

v: Velocidade

Δ_l: Deslocamento

Δ_t: Tempo de duração da ação

Fonte: Borges (2015).

No Quadro 1, Loop *i* representa um ciclo completo e Loop *j* representa o processo iterativo para encontrar um ponto de aceleração aplicável.

Tabela 4 - Exemplo de cálculos para deslocar um trem por 100 metros

$\boldsymbol{pm}$	$\boldsymbol{v}$	$\boldsymbol{R_t}$	$\boldsymbol{F_{ac}}$	$\boldsymbol{F_t}$	Δ_l	Δ_t	$\boldsymbol{C}$
6	18,43	7.738,25	3.879,05	11.617,31	0,02	3,88	0,196
7	18,60	7.740,17	3.780,69	11.520,86	0,02	3,85	0,194
8	18,74	7.749,91	8.838,35	16.588,26	0,02	3,70	0,279
9	20,16	7.768,28	12.927,08	20.695,36	0,02	3,44	0,355
10	21,63	7.785,44	11.680,21	19.465,64	0,02	3,24	0,333
		Total			**0,10**	**18,11**	**1,357**

Fonte: Borges (2015).

Utilizando os dados apresentados na Tabela 1, obtemos com a realização dos cálculos os valores de deslocamento final $\Delta l_{final} = 0{,}10km$, tempo decorrido $\Delta t_{final} = 18{,}11s$ e consumo total $C_{final} = 1{,}357l$, sendo este último calculado a partir do produto $W \times \Delta l_{final}$. A velocidade do trem no ponto de medida 11 é $v = 21{,}63kh/h$.

2.3 JADE - JAVA AGENT DEVELOPMENT FRAMEWORK

O sucesso de um SMA está atrelado na disponibilidade de tecnologias apropriadas (como linguagem de programação, bibliotecas e ferramentas de desenvolvimento) que permitam a implementação dos conceitos e técnicas que formam a base de SMA (BELLIFEMINE; CAIRE; GREENWOOD, 2007). A utilização destas tecnologias torna a implementação de SMA acessível e robusta, com a capacidade de propor soluções que atendem os mais variados problemas e modelagens de sistemas que se utilizam de agentes (CASTRO, 2015). Uma plataforma que se utiliza destes conceitos para desenvolvimento de SMA é JADE (*Java Agent DEvelopment Framework*).

JADE é um *software* livre distribuído pela empresa Telecom Italia (TELECOM ITALIA, 2019) com o objetivo de facilitar o desenvolvimento de aplicativos completos baseados em agentes por meio de um ambiente. Ele implementa os recursos de suporte de ciclo de vida exigidos pelos agentes, a lógica central dos agentes e um conjunto avançado de ferramentas gráficas (BELLIFEMINE; CAIRE; GREENWOOD, 2007). JADE também simplifica a implementação de SMA através de um *middleware* que atende as especificações FIPA, além das ferramentas gráficas que dão suporte as fases de depuração e implantação (TELECOM ITALIA, 2019).

O *framework* foi totalmente desenvolvido em linguagem Java por se beneficiar da grande quantidade de recursos da linguagem, como por exemplo a possibilidade de utilização de bibliotecas de terceiros. Assim ele 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 (BELLIFEMINE; CAIRE; GREENWOOD, 2007).

Em Programação Orientada a Objetos (POO), objeto é a instanciação de uma classe, descrição de atributos e comportamentos a serem aplicados a esse objeto. Ao criar um objeto, este adquire um espaço de memória para armazenamento de seu estado – valores de seus respectivos atributos – além das ações que a ele podem ser aplicadas – conjunto de métodos definidos (RICARTE, 2001). De forma análoga,

agentes também devem ser capazes de armazenar estados – informações sobre o ambiente – como também aplicar ações sobre o meio que estão inseridos.

2.3.1 Comportamentos

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 (BELLIFEMINE; CAIRE; GREENWOOD, 2007).

Um agente pode executar vários comportamentos de forma simultânea, porém estes comportamentos não são executados todos ao mesmo tempo (como threads em JAVA) 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 (BELLIFEMINE; CAIRE; GREENWOOD, 2007).

Os comportamentos podem ser de três tipos (BELLIFEMINE; CAIRE; GREENWOOD, 2007):

- *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* executada toda vez que for chamada até que a execução do agente seja finalizada. Para utilizar este comportamento é necessário estender a classe *CyclicBehaviour*;
- *Generic*: tem como finalidade de, a partir de um valor de status, executar diferentes tarefas de acordo com esse status. Sua execução é concluída quando determinada condição é atendida. Para utilizar este comportamento é necessário estender a classe *Behaviour*.

2.3.2 Implementação de um Agente

Para utilizar JADE no desenvolvimento de SMAs, basta adicionar o *framework* no projeto e importar as classes necessárias. Para a criação de um agente, é necessário estender a classe do tipo *Agent* e importar suas funções. Também, é necessário adicionar os comportamentos que o agente deve adotar. A Figura 3 mostra um exemplo de como inicializar um agente utilizando JADE.

Figura 3 - Criação de um agente em JADE

```
1  import jade.core.Agent;
2
3  public class Agente extends Agent {
4      @Override
5      protected void setup(){
6          System.out.println("Olá, meu nome é "
7              + getLocalName());
8      }
9
10     addBehaviour(new Comportamento());
11 }
```

Fonte: Bellifemine, Caire e Greenwood (2007).

Na linha 3, é feito a importação da classe *Agent*. Na linha 5, é implementado a função *setup* onde é inserido todas as tarefas necessárias. Na linha 7, é impresso na tela o nome do agente criado. Na linha 10, é atrelado o comportamento ao agente criado.

Com o intuito de organizar e padronizar algumas regras para implementação de *softwares* baseados em agentes, foi elaborado padronizações de cunho internacional para desenvolvimento da arquitetura destes sistemas, o qual JADE toma como base.

2.3.3 FIPA

A Fundação para Agentes Físicos Inteligentes, do inglês “*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” (FIPA, 2019). Originalmente criada na Suíça

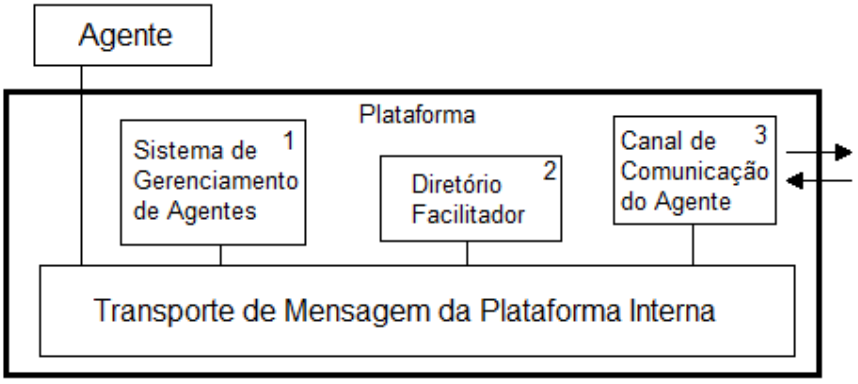
em 1996, foi formada para produzir especificações de padrões de *software* para agentes e assim promover o sucesso no desenvolvimento de aplicações, serviços e equipamentos baseados em agentes (SUGURI, 1999).

Seus primeiros membros, um grupo formado por pesquisadores e empresários, desenvolveram uma coleção de regras que orientam o desenvolvimento de especificações padrão para tecnologias de agentes. Nessa época, agentes já eram conhecidos no meio acadêmico, mas não recebia grande atenção por parte das empresas. Após essa união, foram desenvolvidas regras as quais permitiria que agentes fossem utilizados em diversas aplicações (BELLIFEMINE; CAIRE; GREENWOOD, 2007).

FIPA produz dois tipos de especificações (SUGURI, 1999): (i) normativa e (ii) informativa. Especificações normativas estabelecem comportamentos a um agente e garantem a interoperabilidade com outros sistemas FIPA. Especificações informativas são destinadas a indústrias, a fim de divulgar orientações sobre o uso de tecnologias com especificação FIPA.

FIPA define regras que permitem um conjunto de agentes existir, agir e ser gerenciados, estabelecendo um modelo de referência lógico para a criação, registro, localização, comunicação, navegação e operação de agentes (SUGURI, 1999). Uma representação deste modelo é apresentada na Figura 4.

Figura 4 - Modelo de referência para plataformas de agentes definido pela FIPA



Fonte: adaptado de Teixeira (2012).

Este modelo se baseia em três agentes principais responsáveis pelo gerenciamento da plataforma: (1) Sistema de Gerenciamento de Agentes, (2) Diretório Facilitador e (3) Canal de Comunicação do Agente (TEIXEIRA, 2012), onde:

- Plataforma (*AP*) – do inglês *Agent Platform* – é a infraestrutura física na qual os agentes são implantados. Consiste nas máquinas, sistemas operacionais ou qualquer outro suporte de *software* adicional. A construção do esboço interno da AP fica a cargo de seu desenvolvedor, não sendo padronizado pela FIPA.
- Sistema de Gerenciamento de Agentes (*AMS*) – do inglês *Agent Management System* – representado pelo quadro (1), é o elemento responsável por gerenciar as operações de uma plataforma, como por exemplo criação e exclusão de agentes.
- Diretório Facilitador (*DF*) – do inglês *Directory Facilitator* – representado pelo quadro (2), é um componente opcional responsável por manter uma lista completa e precisa de agentes presentes no meio, assim como fornecer informações atualizadas sobre agentes os quais *DF* tem informações para outros agentes do ambiente.
- Canal de Comunicação do Agente (*ACC*) – do inglês *Agent Communication Channel* – representado pelo quadro (3), é o componente responsável pela troca de mensagens entre agentes. A comunicação entre agentes é realizada utilizando Linguagem de Comunicação de Agentes (*ACL*) – do inglês *Agent Communication Language*.

2.3.3.1Comunicação

Dentro de FIPA é especificada a linguagem para comunicação entre agentes. Denominada FIPA-ACL, esta linguagem é fundamental para a comunicação entre os agentes. As mensagens podem recursivamente conter outras mensagens como contexto e devem conter parâmetros como tipo da mensagem, código de identificação e nome do agente remetente e do agente destinatário (BELLIFEMINE; CAIRE; GREENWOOD, 2007).

FIPA-ACL 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. O Quadro 2 apresenta os parâmetros da linguagem FIPA-ACL utilizados neste trabalho.

Quadro 2 - Parâmetros da linguagem FIPA-ACL

Parâmetro	Descrição
performative	Tipo da mensagem a ser enviada
sender	Identificação do remetente da mensagem
receiver	Identificação do destinatário da mensagem
reply-with	expressão utilizada por um agente para identificação e validação da mensagem
content	conteúdo da mensagem
conversation-id	Identificação única de uma cadeia de mensagens

Fonte: adaptado de Bellifemine, Caire e Greenwood (2007).

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. Cada uma dessas ações define um tipo de resultado esperado. Por exemplo, uma mensagem do tipo *AGREE* indica ao agente destinatário que agente remetente aceitou sua solicitação para realizar determinada ação, a qual provavelmente será realizada no futuro. O descreve os principais tipos de ações. Para maiores detalhes analisar Bellifemine, Caire e Greenwood (2007).

Quadro 3 - Tipos de ações a serem executadas

Tipo da mensagem	Definição
AGREE	Ação de concordar em executar certa tarefa, possivelmente no futuro
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

Fonte: adaptado de Bellifemine, Caire e Greenwood (2007).

Quando enviada uma mensagem do tipo *REQUEST*, é importante que o agente destinatário responda a esta solicitação, deixando o agente remetente ciente que a ação será executada.

2.3.3.2 Arquitetura

JADE é composto por *Containers* de agentes que podem estar distribuídos pela rede. C*ontainer* é o agrupamento de agentes e representa o processo em Java o qual prove o ambiente de execução de JADE e todos os serviços necessários para hospedagem e execução de agentes. O conjunto de C*ontainers* denomina uma AP (TEIXEIRA, 2012).

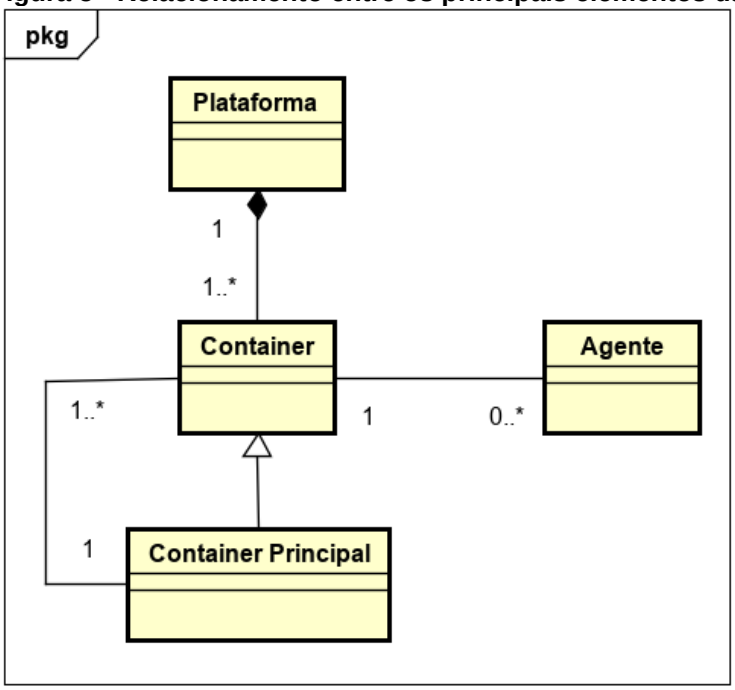
Entre os *Containers*, há um especial denominado C*ontainer Principal*, o qual representa o ponto inicial da plataforma. C*ontainer Principal* é o primeiro *Container* a ser carregado, devendo todos os outros C*ontainers* comunicarem-se com ele. A Figura 5 representa um diagrama de classe com os relacionamentos entre os principais elementos que compõem o sistema (BELLIFEMINE; CAIRE; GREENWOOD, 2007).

O sistema identifica cada *Container* de forma única através de nomes lógicos. Por padrão, *Container Principal* é nomeado como "*Main Container*", enquanto os

outros elementos são nomeados em sequência (“*Container*-1”, “*Container*-2”, ..., “*Container-n*”).

Ao ser carregado, *Container Principal* deve realizar três ações: (i) gerenciar a Tabela de Descritores do Agente Global (*GADT*) – do inglês *Global Agent Descriptor Table*, registro de todos os agentes presentes na Plataforma, incluindo seus estados e localizações, (ii) gerenciar a Tabela de *Container* (*CT*) – do inglês *Container Table*, registro das referências de objetos e endereços de transportes de todos os nós de *Containers* que compõem a Plataforma, e (iii) armazenar os três agentes especiais definidos no modelo de referência da FIPA (*AMS*, *DF* e *ACC*) (TEIXEIRA, 2012).

Figura 5 - Relacionamento entre os principais elementos da arquitetura



Fonte: adaptado de Bellifemine, Caire e Greenwood (2007).

Container Principal fornece para cada *Container* existente uma cópia de GADT para que as informações sejam administradas localmente, a fim de evitar gargalos no *Container Principal*. Quando um *Container* precisa descobrir o local onde se encontra o destinatário de uma mensagem, o mesmo procura primeiramente na sua GADT e, somente em caso de falha, é realizado contato com *Container Principal* a fim de atualizar as informações de sua tabela local, armazenando uma nova cópia para utilizações futuras (BELLIFEMINE; CAIRE; GREENWOOD, 2007).

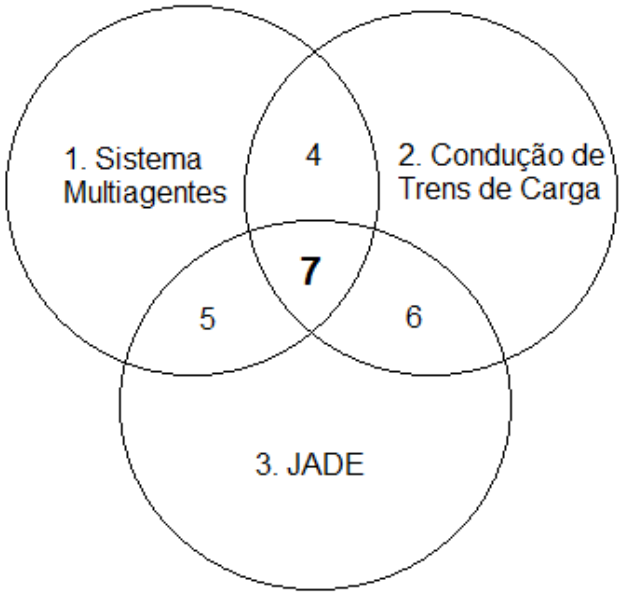
Esta falha se dá principalmente pelo fato do ambiente ser dinâmico, ocorrendo inclusão e exclusão de agentes ou até mesmo a transferência de agentes para outras

plataformas. Nestes casos, ao utilizar um valor obsoleto armazenado o *Container* receberá uma exceção, sendo forçado a atualizar seus dados locais em relação a *Container Principal*.

2.4 TRABALHOS RELACIONADOS

Para facilitar a demonstração do escopo do trabalho, foi criado um diagrama de Venn que representa as áreas abrangidas pelo trabalho a ser desenvolvido, como mostrado na Figura 6. As áreas indicadas foram selecionadas pelo fato de o trabalho realizado utilizar importantes conceitos estudados nas mesmas.

Figura 6 - Diagrama de Venn das áreas abrangidas



Fonte: Autoria própria.

No campo de SMA (1) encontra-se o trabalho realizado por (CUSTODIO, 2017), o qual implementa um agente condutor de veículos com o objetivo de desviar de objetivos estáticos. O agente condutor de veículos é inserido em um ambiente onde há obstáculos definidos dentro de faixas, com o agente circulando entre essas faixas para desviar desses obstáculos. A simulação do ambiente é feita a partir do simulador *Simbad*, o qual é responsável por trocar mensagens com o agente condutor a fim de permitir ao mesmo controle sua representação dentro do simulador. Esta troca de mensagens se dá através de protocolo pré-definido, permitindo ao agente condutor

receber informações sobre o ambiente e enviar as ações a serem realizadas para sua representação dentro do simulador.

No campo de condução de trens de carga (2) encontra-se o trabalho de (BORGES, 2009), o qual propõe a descoberta de padrões com o intuito de auxiliar o maquinista na condução de um trem de carga. A partir da aplicação de técnicas de aprendizagem de máquina e descoberta de conhecimento através de um conjunto de dados de viagens, é definido uma metodologia para se obter padrões que propiciem uma condução segura e econômica. A avaliação da metodologia gerada se deu pela execução destes padrões através de um simulador para selecionar ações a serem aplicadas durante a condução de um trem, assim como a comparação de similaridade entre condução real e condução assistida.

No campo (2) encontra-se também o trabalho de (SATO et al., 2012), o qual apresenta uma abordagem para gerar planos de direção de trem utilizando Planejamento Baseado em Casos (PBC). Cada plano é constituído por um conjunto de ações elaboradas sem intervenção humana e quando aplicadas podem mover um trem em um trecho de ferrovia. Estas ações são planejadas a partir da reutilização e compartilhamento de experiências passadas, tornando-se uma tarefa complexa por fatores como peso, quantidade de locomotivas, quantidade de vagões, perfis de trechos percorridos e condições ambientais. A abordagem apresenta no trabalho é avaliada a partir de métricas como precisão da tarefa de recuperação de caso, eficiência da adaptação da tarefa, eficiência da aplicação destes casos em ambientes realistas e consumo de combustível.

Também em (2) encontra-se o livro de (BRINA, 1983), onde é contemplado o estudo de estradas de ferro. O autor apresenta em sua obra, além de definições sobre todos componentes que constituem um modal férreo, todos os cálculos necessários para a realização de conduções em uma via férrea.

Ainda em (2) encontra-se o trabalho de (STREISKY, 2019), onde é utilizado a técnica de Raciocínio Baseado em Casos com a finalidade de auxiliar na condução de trens de carga. Focado na etapa de recuperação de dados similares de RBC, é utilizado neste trabalho algoritmos de recuperação para selecionar casos passados que mais se assemelham com determinado problema e suas soluções, sendo cada caso correspondente a um conjunto de características referentes ao deslocamento de

um trem. Com a implementação deste trabalho, espera-se aprimorar a condução de trens de carga alcançando principalmente a diminuição no consumo de combustível.

Por fim, encontra-se em (2) o trabalho de (BERNARDO, 2019), o qual apresenta a implementação e aplicação de um algoritmo baseado no comportamento das colônias de formigas no problema da condução de trens de carga. Tomando como base o comportamento eficiente de uma formiga pela busca de recursos, a aplicação deste algoritmo no problema da condução de trens de carga tem por intuito aprimorar a condução destes a partir da seleção de um ponto de aceleração ideal em determinado ponto de medida. A utilização do algoritmo resultou em uma taxa de similaridade alta comparada a condução de um maquinista real como também na diminuição do consumo de combustível e tempo total de viagem.

No campo de JADE (3) encontra-se o trabalho de (TEIXEIRA, 2012), o qual apresenta o funcionamento do *framework* JADE. Desenvolvido com o intuito de facilitar a criação de sistemas baseados em agentes, *Java Agent DEvelopment Framework* provê um ambiente em conformidade com as especificações da FIPA – organização que rege a padronização de sistemas baseados em agentes. JADE prove suporte a implementação destes sistemas através de um modelo de agente programável e extensível, além de um conjunto de ferramentas de gerenciamento e testes.

Ainda em (3) encontra-se o livro de (BELLIFEMINE; CAIRE; GREENWOOD, 2007) o qual apresenta um manual de instruções sobre JADE. Os autores apresentam em sua obra como surgiu JADE, definição da arquitetura FIPA, estrutura do *framework*, funções disponíveis para serem utilizadas na implementação de agentes, além de apresentar um passo a passo de um exemplo prático para desenvolvimento de um SMA vendedor de livros. No exemplo abordado é implementado um agente comprador e vários agentes vendedores, onde o agente comprador procura por determinada literatura e os agentes vendedores respondem com suas respectivas ofertas. Agente comprador fecha negócio com o agente vendedor que oferecer a melhor oferta.

Na junção dos campos de SMA e condução de trens de carga (4) encontra-se o trabalho realizado por (SILVA, 2011), o qual apresenta um sistema computacional inteligente para a condução de trens interurbanos de carga. Este sistema é dotado de um módulo de raciocínio prático para gerar um plano de condução que satisfaça metas

complexas, onde a busca pelo alcance destas metas se dá em duas etapas, sendo elas o (i) planejamento de ações que permitem mover um trem de carga de um ponto a outro, preocupando-se com variáveis como tempo de viagem e economia, e (ii) execução deste plano gerado, modificando assim o ambiente.

Na junção dos campos de SMA e JADE (5) encontra-se o trabalho realizado por (SILVA; MENDES NETO; JÁCOME JÚNIOR, 2011), o qual apresenta uma abordagem baseada em agentes para recomendação sensível ao contexto de objetos de aprendizagem com a finalidade de aperfeiçoar o processo de ensino na aprendizagem móvel. Para se desenvolver ambientes de aprendizado que sejam sensíveis ao contexto do estudante, é essencial que os conteúdos educacionais sejam produzidos de forma padronizada. Uma maneira de padronizar os conteúdos educacionais é através do uso de Objetos de Aprendizagem, que consistem em pequenas unidades de conteúdo que podem ser utilizadas durante o processo de aprendizagem. Utilizando JADE, foi desenvolvido um ambiente de aprendizado móvel que, utilizando-se de objetos de aprendizado e agentes, se adequa as necessidades do estudante de acordo com características do contexto no qual se encontra.

Na junção dos campos de condução de trens de carga e JADE (6) não se encontra nenhum trabalho, pois os trabalhos que contemplam essas áreas de estudo contemplam também o campo de SMA.

O trabalho aqui desenvolvido encontra-se no espaço **7** (cf. Figura 6), pois abrange as três áreas descritas no diagrama. Neste espaço destaca-se o trabalho realizado por (DORDAL et al., 2013), o qual apresenta um sistema inteligente baseado em uma definição de tempo dinâmica, a qual coordena o processo de ultrapassagens de trens viajando em uma mesma seção de uma linha férrea, através de um cruzamento. Cada trem é representado por um agente capaz de tomar decisões baseado em sua posição na linha férrea e em suas ações pré-programadas. A meta dos agentes é trocarem mensagens durante a condução objetivando evitar que dois ou mais trens trafeguem pela mesma seção da linha férrea ao mesmo tempo, evitando assim paradas desnecessárias. O principal objetivo do sistema é auxiliar no gerenciamento dos trens, com foco na redução do consumo de combustível e tempo de viagem.

Estes trabalhos em questão permitem entender melhor principais técnicas das áreas abrangidas a serem utilizadas para desenvolvimento deste trabalho, a partir do

estudo e entendimento da aplicação destas técnicas para solução de problemas do mundo real.

2.5 CONSIDERAÇÕES FINAIS

Neste capítulo foram apresentados os principais conceitos sobre sistemas multiagentes, condução assistida e *framework* JADE. Estes conceitos apresentados são de total importância para entendimento e desenvolvimento do escopo proposto neste trabalho.

No capítulo seguinte será apresentado o desenvolvimento do SMA proposto, apresentando os arquivos necessários a serem carregados, descrição das tarefas de cada agente que constitui o sistema, assim como os passos para implementação de cada um destes.

3 DESENVOLVIMENTO

Para o desenvolvimento do SMA foram utilizados dois tipos de agentes: um condutor e um controlador. O agente condutor representa o condutor de composição – conjunto de locomotivas e vagões – e apresenta funções as quais o permite simular a condução de um trem de carga, carregando informações da composição e a via onde será realizada a condução. O agente controlador representa a estação férrea e apresenta funções o qual permite orientar os agentes condutores presentes na via, como a posição de cada um e o envio de mensagem liberando determinado condutor a seguir viagem.

3.1 DADOS DE CONDUÇÃO

Para simulação do trem a ser utilizado e da via a ser percorrida por parte do agente condutor, é realizado a leitura de arquivos no formato XML. Elaborados por Borges (2015), estes arquivos contêm dados fundamentais para a simulação fiel ao mundo real da ação de conduzir um trem de carga.

3.1.1 Via

Os dados da via a ser percorrida são adquiridos a partir de arquivo XML gerados pela conversão de planos cartográficos disponibilizados por uma companhia do setor férreo. Este arquivo é o mesmo utilizado em Borges (2009 e 2015), e contém os seguintes dados: identificador da via, a velocidade média da via, a distância entre os pontos de medição – utilizados para os cálculos de condução e a listagem dos pontos de medida. Os dados utilizados sobre cada ponto de medida são o código de identificação do ponto de medida, velocidade máxima permitida no trecho analisado e o quilometro da via que o trem se encontra. A Figura 7 mostra um exemplo da representação dos dados da via em formato XML.

Figura 7 - Representação dos dados de uma via em formato XML

```xml
<viaFerrea>
    <identificador>Exemplo via ferrea</identificador>
    <distanciaPonto>20</distanciaPonto>
    <bitolaLinha>1.60</bitolaLinha>
    <velocidadeMedia>100</velocidadeMedia>
    <listaDePontosDeMedida>
                <pontoDeMedida>
            <id>1</id>
            <velocidadeMax>60</velocidadeMax>
            <km>204</km>
            <rampa ini="0.0" fim="20.0">-0.98642856</rampa>
            <raioCurva ini="0.0" fim="20.0">0.0</raioCurva>
            <ac ini="0.0" fim="20.0">0.0</ac>
            <g20 ini="0.0" fim="20.0">0.0</g20>
            <altitude ini="0.0" fim="20.0">527.573</altitude>
            <localizacao>
                <latitude>0</latitude>
                <longitude>0</longitude>
            </localizacao>
        </pontoDeMedida>
        <pontoDeMedida>
            <id>2</id>
            <velocidadeMax>60</velocidadeMax>
            <km>204</km>
            <rampa ini="0.0" fim="20.0">-0.98642856</rampa>
            <raioCurva ini="0.0" fim="20.0">0.0</raioCurva>
            <ac ini="0.0" fim="20.0">0.0</ac>
            <g20 ini="0.0" fim="20.0">0.0</g20>
            <altitude ini="0.0" fim="20.0">527.573</altitude>
            <localizacao>
                <latitude>0</latitude>
                <longitude>0</longitude>
            </localizacao>
        </pontoDeMedida>
        </listaDePontosDeMedida>
</viaFerrea>
```

Fonte: Autoria própria.

Os dados da via são carregados antes da inicialização do agente condutor, conforme ilustra o Código 1. Os parâmetros necessários para carregar a via corretamente são o caminho onde se encontra o arquivo XML e o número da viagem, parâmetro este que indicará a distância da viagem.

```
String caminhoArquivoVia = "\via01.xml";
String viagemNro = "01";

CIVia = new CIVia(caminhoArquivoVia,
    Integer.parseInt(viagemNro));
```

Código 1 - Função para iniciar via

A classe *CIVia* é responsável, principalmente, pela criação dos objetos que representam a via utilizada pelos agentes, onde cada característica da via é mapeada

para um objeto. Após a criação dos objetos, todos os dados são repassados aos agentes no momento de sua inicialização.

3.1.2 Composição

Ao conjunto de locomotivas e vagões que formam um trem dá-se o nome de composição. Os dados sobre a composição a ser utilizada são adquiridos a partir da leitura do arquivo XML, sendo cada arquivo a representação de uma composição de diferentes configurações. No arquivo XML referente a composição, encontra-se dados referentes a locomotiva e vagões. A Figura 8 mostra um exemplo da representação dos dados de uma composição em formato XML.

Figura 8 - Representação dos dados de uma composição em formato XML

```
<composicao id="Composicao de Teste">
    <distanciaEntreVeiculos>1.25</distanciaEntreVeiculos>
    <listaVeiculos>
        <locomotiva quantidade="3">
            <veiculo>
                <tara>169.7</tara>
                <numeroDeEixo>4</numeroDeEixo>
                <areaFrontal>120</areaFrontal>
                <comprimento>20</comprimento>
            </veiculo>
            <propulsor>
                <modelo>C30</modelo>
                <quantidadeDePontos>11</quantidadeDePontos>
                <idPontoAceleracaoPartida>0</idPontoAceleracaoPartida>
                <pontoDeAceleracao>
                    <id>0</id>
                    <potencia>0</potencia>
                    <consumo>0.3168</consumo>
                </pontoDeAceleracao>
                <pontoDeAceleracao>
                    <id>1</id>
                    <potencia>100</potencia>
                    <consumo>0.5670</consumo>
                </pontoDeAceleracao>
            </propulsor>
        </locomotiva>
        <vagao quantidade="58">
            <veiculo>
                <tara>29.46</tara>
                <numeroDeEixo>4</numeroDeEixo>
                <areaFrontal>120</areaFrontal>
                <comprimento>20</comprimento>
            </veiculo>
            <carga>70</carga>
        </vagao>
```

Fonte: Autoria própria.

Os dados sobre locomotiva disponíveis são a quantidade de locomotivas utilizadas na composição, dados do veículo e dados do propulsor utilizado na

locomotiva. Os dados disponíveis sobre o veículo são o peso da locomotiva em toneladas, número de eixos, área frontal em metros quadrados e comprimento da locomotiva em metros. Os dados disponíveis sobre o propulsor são o modelo do propulsor, a quantidade de pontos de aceleração disponíveis, o ponto de aceleração aplicado ao se dar partida na locomotiva e dados sobre cada ponto de aceleração presente, sendo estes o número do ponto de aceleração, potência gerada em *HP* (*Horse Power*) e consumo em Litros.

Os dados sobre vagão disponíveis são: a quantidade de vagões utilizadas na composição, quantidade de carga máxima por vagão em toneladas e dados do veículo, sendo estes o peso do vagão em toneladas, número de eixos, área frontal em metros quadrados e comprimento em metros.

O Código 2 mostra a função de inicialização de uma composição. Os parâmetros necessários para carregar uma composição são o número da viagem – parâmetro o qual dirá a distância da viagem, o caminho onde se encontra o arquivo XML, a variável onde foi carregada as informações da via, o tipo de viagem, o número de tentativas realizadas e a posição da estação.

```
1  String caminhoArqComposicao = "\composicao-lap01.xml";
2  String tipoViagem = "";
3  int nroTentativas = 1;
4  double posicaoEstacao = 0;
5
6  CIComposicao composicao = new CIComposicao(viagemNro,
7      caminhoArqComposicao, via, tipoViagem, nroTentativas, posicaoEstacao);
```

Código 2 - Função para iniciar composição

Internamente, o construtor *CIComposicao* é responsável por ler o arquivo XML da composição, instanciando a quantidade de objetos *Locomotiva* e *Vagao* especificados no arquivo XML. Maiores detalhes podem ser obtidos em Borges (2009 e 2015). Após a composição ser criada e os dados carregados, o objeto composição é enviado ao agente condutor. Este utilizará as informações da composição para os cálculos matemáticos de deslocamento, bem como para validações das ações.

3.2 AGENTE CONTROLADOR

Em analogia ao mundo real, o agente controlador representa a estação férrea e é responsável por orientar o posicionamento dos agentes condutores na via. O agente mantém, internamente, uma lista atualizada com a posição dos agentes condutores, evitando assim colisões entre esses. Além disso, ele libera os agentes condutores a seguir em viagem baseado em uma distância de segurança entre o agente condutor na espera para seguir viagem e o agente condutor a sua frente.

O agente Controlador mantém a lista atualizada da posição de cada agente condutor, além do último condutor a iniciar sua viagem. Após liberar o primeiro condutor presente na lista a seguir viagem, agente controlador envia uma mensagem a todos os condutores requisitando suas posições. O Código 3 descreve a requisição de posicionamento dos agentes condutores.

```
1   str = "localizacao";
2   mensagem = new ACLMessage(ACLMessage.REQUEST);
3
4   for (int i = 0; i < condutores.size(); i++) {
5       mensagem.addReceiver(condutores.get(i).getCondutor());
6   }
7
8   mensagem.setContent(str);
9   mensagem.setConversationId(str);
10  mensagem.setReplyWith(str + System.currentTimeMillis());
11
12  myAgent.send(mensagem);
```

Código 3 - Mensagem de solicitação de posicionamento

Nas linhas 1 e 2 são configurados o texto e o tipo da mensagem a ser enviada, respectivamente. Nas linhas 4 a 6 é adicionado como destinatário todos os agentes do tipo condutor presentes na via. Nas linhas 8 a 10 é configurado o corpo, a identificação e o padrão de resposta da mensagem, respectivamente. Na linha 12, a mensagem é enviada.

Quando o último agente a seguir viagem alcançar determinada distância percorrida, controlador permite que o próximo condutor siga em viagem, mantendo assim uma distância de segurança entre as composições, conforme ilustra o Código 4.

Na linha 1 é verificado se o trem que está na fila pode prosseguir com sua viagem. Nas linhas 2 e 3 são configurados o texto e o tipo da mensagem a ser enviada, respectivamente. Nas linhas 5 a 8 são configurados o destinatário, o conteúdo, a identificação e o padrão de resposta da mensagem, respectivamente. Na linha 10, a mensagem é enviada ao respectivo condutor.

```
1  if (prosseguir == true) {
2      str = "prosseguir";
3      mensagem = new ACLMessage(ACLMessage.INFORM);
4
5      mensagem.addReceiver(condutores.get(condutorCount).getCondutor());
6      mensagem.setContent(str);
7      mensagem.setConversationId(str);
8      mensagem.setReplyWith(str + System.currentTimeMillis());
9
10     myAgent.send(mensagem);
11 }
```

Código 4 - Código de controle de liberação de trens

Ao ser inicializado, como ilustra o Código 5, agente controlador realiza seu cadastro no *Directory Facilitator* (*DF*) com a descrição “AgenteControlador” e em seguida são adicionados seus comportamentos.

```
1  @Override
2  protected void setup() {
3      try {
4          String nome = "AgenteControlador";
5          DFAgentDescription dfa = new DFAgentDescription();
6          ServiceDescription sd = new ServiceDescription();
7
8          dfa.setName(getAID());
9          sd.setType(nome);
10         sd.setName(nome);
11         dfa.addServices(sd);
12
13         DFService.register(this, dfa);
14     } catch (FIPAException ex) {
15         Logger.getLogger(getClass().getName())
16             .log(Level.SEVERE, null, ex);
17     }
18
19     addBehaviour(new AgenteControladorBehaviour());
20 }
```

Código 5 - Inicialização do Agente Controlador

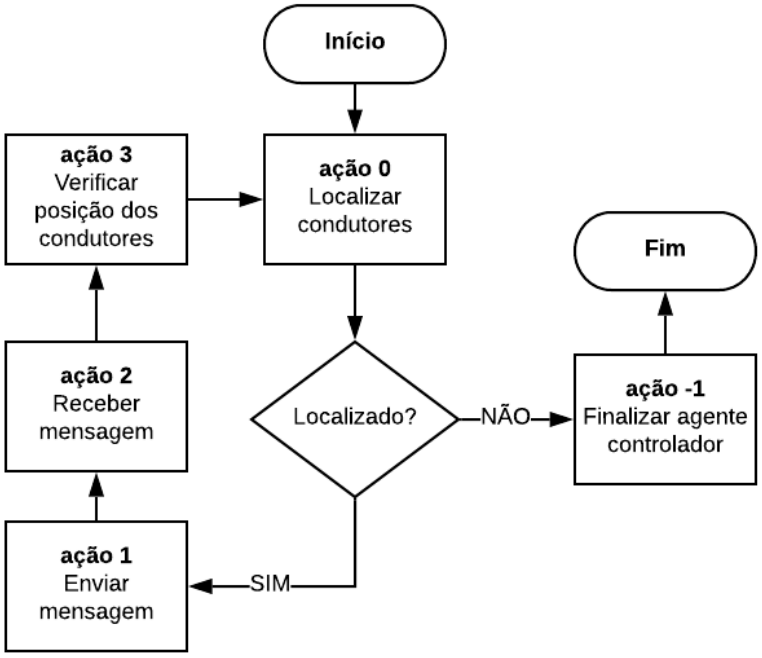
Nas linhas 4 a 11 são configuradas informações sobre o agente a ser cadastrado: seu nome, tipo e nome do serviço prestado. Na linha 13, agente

controlador é registrado ao *DF*. Na linha 19 os comportamentos são adicionados ao agente controlador.

Agente controlador apresenta como comportamentos a busca por agentes do tipo “AgenteCondutor” cadastrados no *DF*, o envio e recebimento de mensagens dos agentes condutores e verificação da distância de segurança entre condutores.

Implementado com comportamento do tipo *GENERIC* (cf. seção 1.1.1), a organização das ações do agente controlador pode ser representada a partir de um fluxograma, como indicado na Figura 9.

Figura 9 - Fluxograma de ações do agente controlador



Fonte: Autoria própria.

As seções a seguir descrevem o funcionamento das ações do agente controlador, conforme diagrama da Figura 9.

3.2.1 Ação 0

Ao iniciar agente controlador, como ilustrado no Código 6, é realizada uma busca em *DF* de todos os agentes do tipo “AgenteCondutor”, mantendo o nome de todos estes em uma lista.

```
1   case 0:
2       nomeServico = "AgenteCondutor";
3       DFAgentDescription[] result = buscarAgentesPorServico(nomeServico);
4
5       if (result.length == 0) {
6           acao = -1;
7           break;
8       }
9
10      condutores.clear();
11      for (DFAgentDescription c : result) {
12          condutores.add(new CondutorDTO(c.getName()));
13      }
14
15      acao = 1;
16      break;
```

Código 6 - Localização de serviços pelo Agente Controlador

Na linha 3, é realizado a busca por agentes do tipo “AgenteCondutor”. Na linha 5, caso não haja retorno da busca, agente controlador é encerrado, visto que todos os condutores chegaram ao destino. Na linha 12 são adicionados todos os valores retornados na busca em uma lista. Esses valores são manipulados a partir de uma classe denominada *CondutorDTO*, como ilustrado no Código 7.

Esta classe é utilizada para facilitar o controle de informações de cada condutor presente na via, sendo eles a identificação do agente condutor, nome designado ao agente condutor, posição atual do condutor na via, distância percorrida, comprimento da composição e tempo de viagem.

```
1  public class CondutorDTO {
2      private AID condutor;
3      private String nomeCondutor;
4      private double kmAtual;
5      private double distanciaPercorrida;
6      private double comprimento;
7      private double tempo;
8
9      public CondutorDTO() {
10     }
11
12     public CondutorDTO(AID condutor){
13         this.condutor = condutor;
14     }
15
16     public CondutorDTO(String nomeCondutor, double kmAtual,
17         double distanciaPercorrida, double comprimento, double tempo){
18         this.nomeCondutor = nomeCondutor;
19         this.kmAtual = kmAtual;
20         this.distanciaPercorrida = distanciaPercorrida;
21         this.comprimento = comprimento;
22         this.tempo = tempo;
23     }
24 }
```

Código 7 - Estrutura da classe *CondutorDTO*

Nas linhas 2 a 7 são declaradas as variáveis onde serão inseridas as respectivas informações do agente condutor. Nas linhas 9 a 22, são declarados os construtores disponíveis da classe *CondutorDTO*.

3.2.2 Ação 1

Em ação 1, ilustrada pelo Código 8, é enviada uma mensagem do tipo *REQUEST* com o conteúdo “localizacao” a todos os condutores requisitando suas respectivas posições. Caso um condutor possa iniciar sua viagem, é enviado ao mesmo uma mensagem do tipo *INFORM* com o conteúdo “prosseguir”. A sequência de liberação de viagem se dá pela ordem em que se encontram os agentes condutores na lista gerada na ação 0.

```
1   case 1:
2       if (prosseguir == true) {
3           str = "prosseguir";
4           mensagem = new ACLMessage(ACLMessage.INFORM);
5
6           mensagem.addReceiver(condutores.get(condutorCount).getCondutor());
7           mensagem.setContent(str);
8           mensagem.setConversationId(str);
9           mensagem.setReplyWith(str + System.currentTimeMillis());
10
11          myAgent.send(mensagem);
12
13          nroMensagens++;
14          condutorCount++;
15      }
16
17      str = "localizacao";
18      mensagem = new ACLMessage(ACLMessage.REQUEST);
19
20      for (int i = 0; i < condutores.size(); i++) {
21          mensagem.addReceiver(condutores.get(i).getCondutor());
22      }
23
24      mensagem.setContent(str);
25      mensagem.setConversationId(str);
26      mensagem.setReplyWith(str + System.currentTimeMillis());
27
28      myAgent.send(mensagem);
29
30      nroMensagens++;
31
32      acao = 2;
33      break;
```

Código 8 - Envio de mensagens aos agentes condutores

Nas linhas 2 a 15, caso a condutor da vez possa iniciar a sua viagem, é enviado a este uma mensagem com o conteúdo “prosseguir” para comunicá-lo que sua viagem está liberada. Nas linhas 17 a 28 é enviado a todos os agentes condutores uma requisição de suas respectivas posições na via. Nas linhas 13 e 30, é incrementado a variável que armazena a quantidade de mensagens trocadas com agente controlador. Ao enviar as mensagens, agente controlador executa ação 2.

3.2.3 Ação 2

Em ação 2, ilustrado pelo Código 9, são recebidas todas as mensagens enviadas por parte dos condutores. Utilizando uma estrutura switch/case para filtragem, as mensagens recebidas podem possuir o conteúdo “prosseguir”, “localizacao” ou “finalizar”. As mensagens com “prosseguir” e “localizacao” trata-se de respostas para requisições enviadas aos condutores. A mensagem com conteúdo

“finalizar” é recebida pelo controlador quando um agente condutor chega ao seu destino.

```
1   case 2:
2      mensagem = myAgent.receive();
3
4      if (mensagem != null) {
5          nroMensagens++;
6          String conversationId = mensagem.getConversationId();
7
8          switch (conversationId) {
9              case "prosseguir":
10                 condutorViajando++;
11                 prosseguir = false;
12
13                 break;
14             case "localizacao":
15                 for (CondutorDTO c : condutores) {
16                     if (c.getCondutor().getLocalName()
17                             .equals(mensagem.getSender().getLocalName())) {
18                         atualizarPosicao(c, mensagem.getContent());
19                         historicoViagem.add(new HistoricoControladorDTO(c, nroMensagens));
20                     }
21                 }
22
23                 break;
24             case "finalizar":
25                 System.out.println("Controlador: condutor "
26                     + mensagem.getSender().getLocalName() + " chegou ao destino");
27                 condutorCount--;
28                 break;
29         }
30     } else {
31         block();
32     }
```

Código 9 - Tratamento de mensagens pelo agente controlador

Ao receber uma mensagem com conteúdo “prosseguir” agente controlador está sendo notificado que o agente condutor que enviou a mensagem iniciou sua viagem. Na linha 10, é incrementado o contador que indica o número de condutores em viagem. Na linha 11, é configurado como falso a variável que indica se o próximo condutor da fila pode iniciar sua condução.

Ao receber uma mensagem com conteúdo “localizacao” agente controlador atualiza os dados do condutor, adicionando estas informações em uma lista com um histórico de ações ocorridas na via. Das linhas 15 a 19, controlador procura em seu registro de condutores o remetente da mensagem, atualiza suas informações e as salva no histórico de viagem, respectivamente.

Ao receber uma mensagem com conteúdo “finalizar”, agente controlador toma ciência de que o condutor remetente chegou ao seu destino. Na linha 25 é mostrado ao usuário que determinado condutor chegou ao destino. Na linha 27 é decrementado o contador que indica o número de condutores em viagem.

Para atualizar os dados da composição a qual enviou a mensagem, é utilizada a função *atualizarPosicao*, ilustrada no Código 10, recebendo como parâmetro os dados do agente que enviou a mensagem e o conteúdo da mensagem

```
1  public void atualizarPosicao(CondutorDTO c, String json) {
2      Gson gson = new Gson();
3      CondutorDTO aux = gson.fromJson(json, CondutorDTO.class);
4
5      c.setKmAtual(aux.getKmAtual());
6      c.setDistanciaPercorrida(aux.getDistanciaPercorrida());
7      c.setComprimento(aux.getComprimento());
8      c.setTempo(aux.getTempo());
9  }
```

Código 10 - Função *atualizarPosicao*

Na linha 3 o conteúdo da mensagem recebida é convertido de formato JSON para a classe *CondutorDTO* (cf. Código 7). Nas linhas 5 a 8, os dados do condutor são atualizados com os novos dados recebidos.

Os valores a serem armazenados no histórico de condutores são manipulados a partir de uma classe denominada *HistoricoControladorDTO*, ilustrada no Código 11. Esta classe é utilizada para facilitar no manuseio dos valores recebidos.

```
1  public class HistoricoControladorDTO {
2      private String condutor;
3      private Double distanciaPercorrida;
4      private Double kmAtual;
5      private Double tempo;
6      private Integer nroMensagens;
7
8      public HistoricoControladorDTO(CondutorDTO dto, Integer nroMensagens) {
9          this.condutor = dto.getCondutor().getLocalName();
10         this.distanciaPercorrida = setarDuasCasas(new BigDecimal(dto
11             .getDistanciaPercorrida())).doubleValue();
12         this.kmAtual = setarDuasCasas(new BigDecimal(dto
13             .getKmAtual())).doubleValue();
14         this.tempo = setarDuasCasas(new BigDecimal(dto
15             .getTempo())).doubleValue();
16         this.nroMensagens = nroMensagens;
17     }
18
19     private static BigDecimal setarDuasCasas(BigDecimal bd) {
20         return bd.setScale(2, BigDecimal.ROUND_HALF_UP);
21     }
22 }
```

Código 11 - Estrutura da classe *HistoricoControladorDTO*

Nas linhas 2 a 6 são declaradas as variáveis onde serão inseridas as respectivas informações a serem salvas. Nas linhas 8 a 16 é declarado o construtor

da classe. Na linha 19 é declarada a função *setarDuasCasas*, responsável por arredondar os valores para duas casas decimais.

3.2.4 Ação 3

Em ação 3, ilustrada no Código 12, agente controlador verifica a posição do último trem em viagem em relação ao condutor que está na vez para iniciar viagem. Se a distância entre os dois agentes analisados for igual ou maior que *dist* (em metros), o agente controlador então libera viagem ao condutor da vez. O valor de *dist* pode ser modificado em função do mecanismo de segurança utilizado pelo controlador.

```
1   case 3:
2       double p = condutores.get(condutorCount).getDistanciaPercorrida()
3           - condutores.get(condutorCount).getComprimento();
4
5       if (p >= dist) {
6           prosseguir = true;
7       }
8
9       acao = 0;
10      break;
```

Código 12 - Verificação da distância de segurança entre condutores

Na linha 2, é calculado a posição relativa do último vagão que constitui a composição sendo analisada. Na linha 5, caso o valor calculado anteriormente seja maior ou igual que *dist*, é configurada como verdadeira a variável que indica se o próximo condutor da lista pode seguir viagem. Com esta variável configurada como verdadeira, uma mensagem será enviada ao condutor na próxima execução da ação 0.

3.3 AGENTE CONDUTOR

Em analogia ao mundo real, agente condutor representa um trem de carga a ser conduzido na via. É possível ter um ou mais agentes condutores dentro de uma

mesma via, sendo importante uma maneira de controle entre eles para que não haja contratempos durante a condução, como colisão entre condutores.

Com intenção de evitar situações adversas entre os condutores presentes no ambiente, estes devem manter constante comunicação com a estação férrea, representado no escopo do problema pelo agente controlador. Os condutores devem iniciar viagem somente ao receber liberação por parte da estação férrea e enviar dados sobre sua posição sempre que solicitado. Ao chegar em seu destino, o condutor deve notificar a estação férrea sobre o fim de sua viagem.

Para que possua a capacidade de conduzir um trem de carga, agente condutor carrega informações sobre a composição e a via a ser utilizada. Com esses dados, é realizada uma série de cálculos afim de obter informações necessárias para a condução, como distância percorrida, velocidade e tempo gasto para percorrer determinada distância. O Código 13 apresenta a inicialização do agente condutor.

```
1   @Override
2   protected void setup() {
3       Object[] args = getArguments();
4
5       if (args != null && args.length > 0) {
6           try {
7               composicao = (CIComposicao) args[0];
8               calc = new Calculo();
9
10              inicializarAgenteCondutor();
11
12              String nome = "AgenteCondutor";
13              DFAgentDescription dfAgentDescription = new DFAgentDescription();
14              ServiceDescription serviceDescription = new ServiceDescription();
15
16              dfAgentDescription.setName(this.getAID());
17              serviceDescription.setType(nome);
18              serviceDescription.setName(nome);
19              dfAgentDescription.addServices(serviceDescription);
20
21              DFService.register(this, dfAgentDescription);
22          } catch (FIPAException ex) {
23              Logger.getLogger(AgenteCondutor.class.getName())
24                  .log(Level.SEVERE, null, ex);
25          }
26
27          addBehaviour(new AgenteCondutorBehaviour(composicao, calc));
28      } else {
29          doDelete();
30      }
31  }
```

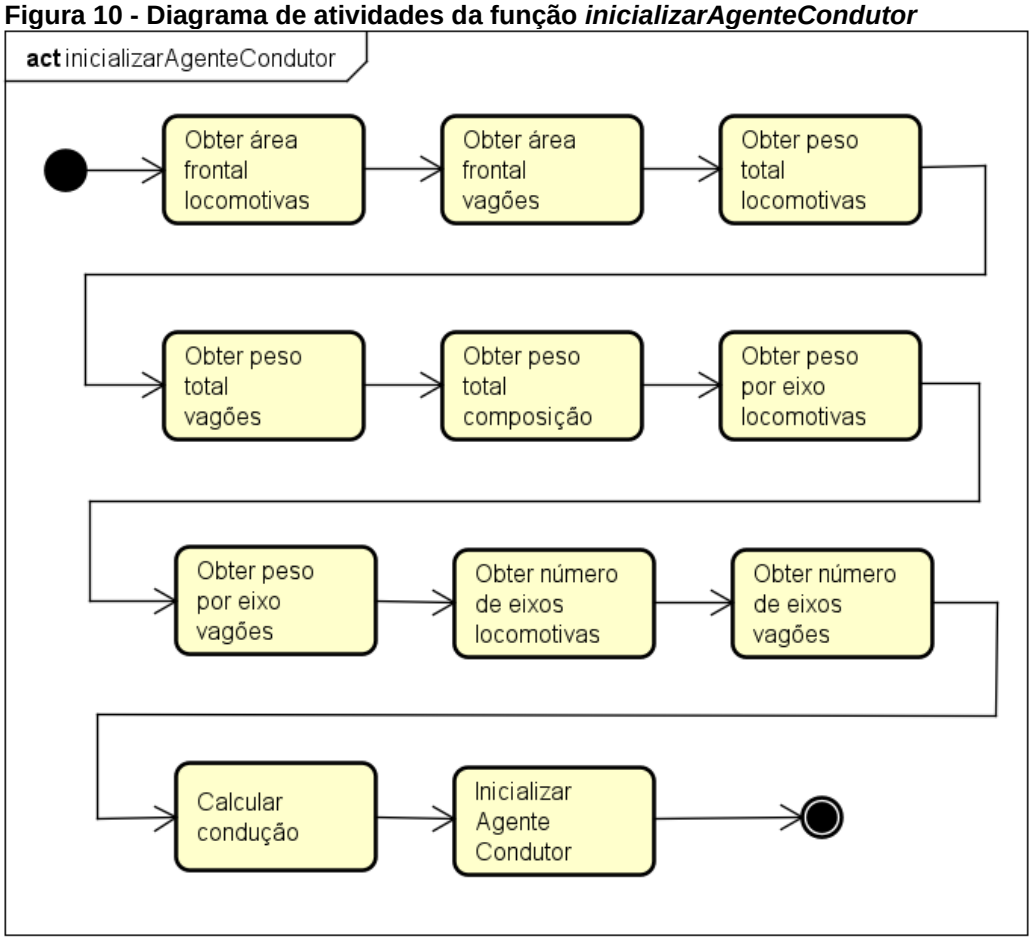
Código 13 - Inicialização do Agente Condutor

Na linha 10 são carregados os dados da composição passados por parâmetro. Se estes dados não forem transmitidos, agente condutor é encerrado chamando a

função na linha 29. Nas linhas 12 e 13 são instanciados os objetos responsáveis pelo agente *DF* e serviço do JADE. Das linhas 16 a 19 são configuradas informações sobre o agente a ser cadastrado. Na linha 21 o serviço prestado pelo agente condutor é registrado no *DF*. Na linha 21, o comportamento é adicionado ao agente condutor.

A função *inicializarAgenteCondutor* é responsável por inicializar as variáveis com as informações recebidas por parâmetro da composição a ser utilizada. Esta função pode ser representada a partir de um diagrama de atividades, como ilustrado na Figura 10.

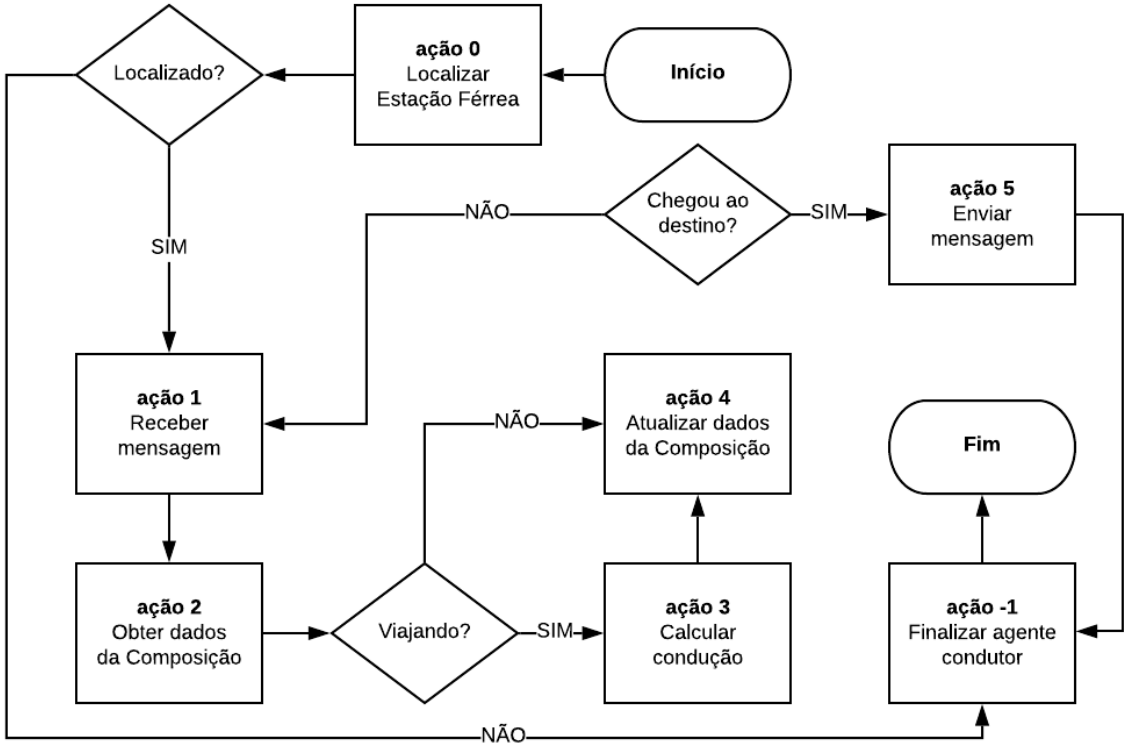
Figura 10 - Diagrama de atividades da função *inicializarAgenteCondutor*



Fonte: Autoria própria.

Implementado com comportamento do tipo *GENERIC*, a organização das ações do agente condutor pode ser representada a partir de um fluxograma, como indicado na Figura 11.

Figura 11 - Fluxograma de ações do agente condutor



Fonte: Autoria própria.

As seções a seguir descrevem o funcionamento das ações do agente condutor, conforme diagrama da Figura 11.

3.3.1 Ação 0

Ao iniciar agente condutor, como ilustrado no Código 14, é feito uma busca em *DF* de todos os agentes que prestam o serviço do tipo “AgenteControlador”, mantendo o nome de todos estes em uma lista.

```
1   case 0:
2       String nomeServico = "AgenteControlador";
3       DFAgentDescription[] result = buscarAgentesPorServico(nomeServico);
4
5       if (result.length == 0) {
6           acao = -1;
7           break;
8       }
9
10      for (DFAgentDescription c : result) {
11          controladores.add(c.getName());
12      }
13
14      acao = 1;
15      break;
```

Código 14 - Busca de agentes pelo agente condutor

Na linha 3, é realizada a busca por agentes do tipo “AgenteControlador”. Na linha 5, caso não haja retorno da busca, agente condutor é encerrado, visto que não há nenhum agente controlador existente. Na linha 11 é adicionado todos os valores retornados na busca em uma lista. Ao término da ação 0 a ação do agente é mudada para 1 (cf. linha 14), descrita a seguir.

3.3.2 Ação 1

Em ação 1, ilustrada no Código 15, agente condutor recebe todas as mensagens enviadas por parte do controlador. Para filtrar as mensagens recebidas, é utilizado uma estrutura switch/case. As mensagens recebidas podem ser do tipo “prosseguir”, “localizacao” e “finalizar”.

As mensagens podem possuir o conteúdo “prosseguir” ou “localizacao”. As mensagens “prosseguir” e “localizacao” são requisições do controlador para que o condutor siga com sua viagem e envie sua localização atual, respectivamente.

```
1   case 1:
2       mensagem = myAgent.receive();
3
4       if (mensagem != null) {
5           String conversationId = mensagem.getConversationId();
6           resposta = mensagem.createReply();
7           nroMensagens++;
8
9           switch (conversationId) {
10              case "prosseguir":
11                  resposta.setContent("viajando");
12                  viajando = true;
13                  break;
14              case "localizacao":
15                  resposta.setContent(gerarLocalizacao());
16                  break;
17          }
18
19          myAgent.send(resposta);
20          acao = 2;
21      } else {
22          block();
23      }
```

Código 15 - Tratamento de mensagens pelo agente condutor

Ao receber uma mensagem do tipo “prosseguir”, agente condutor envia uma resposta a solicitação do controlador de que iniciou sua viagem. Na linha 11, é enviada uma resposta a requisição recebida do controlador. Na linha 12, é configurado como verdadeiro a variável que indica que o condutor está em viagem.

Ao receber uma mensagem “localizacao”, agente condutor envia uma resposta a solicitação do controlador com as informações sobre sua posição atual. Na linha 15, é enviada uma resposta a requisição recebida do controlador com os dados da composição.

O conteúdo da mensagem enviada na linha 15 é gerada pela função *gerarLocalizacao*, como ilustrada no Código 16.

```
1  public String gerarLocalizacao() {
2      Gson gson = new Gson();
3
4      return gson.toJson(new CondutorDTO
5          (myAgent.getAID().getLocalName(),
6              kmAtual, distanciaPercorrida,
7              comprimento, tempoViagem));
8  }
```

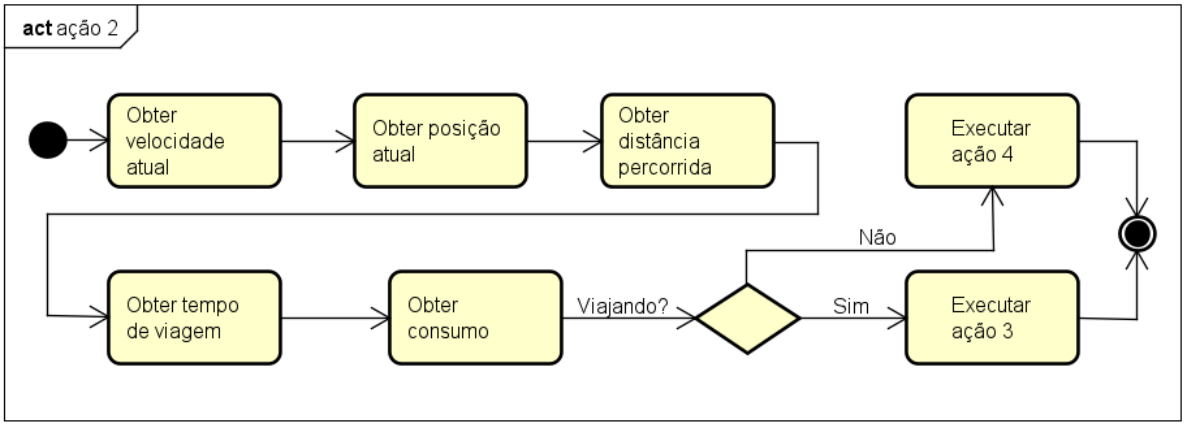
Código 16 - Função *gerarLocalizacao*

Na linha 4 é retornado um JSON com os valores da composição. Os dados presentes no pacote são o nome do agente condutor, quilometro atual na via, distância percorrida, comprimento da composição e tempo total de viagem. Estes dados são geridos a partir da classe *CondutorDTO* (cf. Código 7).

3.3.3 Ação 2

Em ação 2 é realizada a leitura de todos os dados necessários da composição, sendo eles velocidade atual, distância total percorrida, consumo de combustível, posição atual da composição na via e tempo total de viagem. Estes dados são salvos em uma lista a fim de manter um histórico de viagem do condutor. A Figura 12 ilustra a ação 2 através de um diagrama de atividades.

Figura 12 - Diagrama de atividades da ação 2



Fonte: Autoria própria.

A próxima ação a ser executada pelo condutor dependerá de seu estado. Se em viagem, agente condutor deve realizar os cálculos para condução executando a ação 3. Caso contrário é necessário que o condutor apenas atualize suas informações

referentes a consumo de combustível e tempo de espera, executando diretamente ação 4.

3.3.4 Ação 3

Em ação 3, ilustrada no Código 17, são executados os cálculos para condução da composição, utilizando os valores recuperados na ação 2. Após a execução dos cálculos, é realizado a validação dos resultados afim de identificar se o ponto de aceleração aplicado é suficiente para movimentar a composição. Os cálculos de condução e as validações aplicadas são detalhados na seção 2.2 deste trabalho.

```
1  case 3:
2      pontoAceleracao = new Random().nextInt(9);
3      aplicarPA();
4      faltaraForca();
5      vaiPatinar();
6
7      acao = 4;
8      break;
```

Código 17 - Aplicação dos cálculos de condução e suas validações

Na linha 2, é selecionado um novo ponto de aceleração. Na linha 3 é executada a função *aplicarPA*, a qual é responsável por aplicar o PA selecionado e realizar os cálculos de condução. Nas linhas 4 e 5 são executadas as funções *faltaraForca* e *vaiPatinar* para verificar se o PA aplicado será efetivo. Na linha 7 a ação do agente é alterada para 4. Estas funções não recebem nenhum parâmetro nem retornam algum valor pelo fato de se utilizarem de variáveis de classe.

O Código 18 ilustra a implementação da função *aplicarPA*. O Código 19 ilustra a implementação da função *faltaraForca*. O Código 20 ilustra a implementação da função *vaiPatinar*.

```
1  public void aplicarPA() {
2      composicao.obterComposicao()
3          .aplicarPontoAceleracao(pontoAceleracao);
4      potencia = composicao.obterComposicao()
5          .obterLocomotivaLider().obterPropulsor()
6          .obterPontoAceleracaoAtual().obterPotencia();
7
8      deslocamento = calculo.calculoMovimento(velocidadeAtual, potencia);
9  }
```

Código 18 - Implementação da função *aplicarPA*

Na linha 2 é aplicado o PA escolhido a composição. Na linha 4 é recuperada a potência gerada pelo PA aplicado, valor este que será utilizado para realização dos cálculos de condução. Na linha 8 é realizado os cálculos de condução (cf. seção 0).

```
1  public boolean faltaraForca() {
2      while (deslocamento.getForcaAceleracao() <= 0.0f) {
3          if (pontoAceleracao < 8) {
4              pontoAceleracao++;
5              aplicarPA();
6          } else {
7              velocidadeAtual -= 0.5f;
8              aplicarPA();
9          }
10     }
11
12     return true;
13 }
```

Código 19 - Implementação da função *faltaraForca*

Enquanto a força de aceleração gerada for menor ou igual a 0, é aplicada um novo ponto de aceleração. Na linha 4, se o PA atual for menor a 8 (maior ponto de aceleração da locomotiva), PA é incrementado e a função *aplicarPA* é novamente aplicada. Na linha 7, se o PA atual for igual a 8, a velocidade atual é decrementada e a função *aplicarPA* é novamente aplicada.

```
1  public boolean vaiPatinar() {
2      double coeficienteAderencia = composicao.obterComposicao()
3          .obterSensor().obterCoeficienteDeAderecia();
4      double esforcoTratorAderente = composicao.obterComposicao()
5          .obterEsforcoTratorAderente(coeficienteAderencia, deslocamento.getVelocidadeMedia());
6      double esforcoTratorEfetivo = composicao.obterComposicao()
7          .obterLocomotivaLider().obterEsforcoTrator(potencia, velocidadeAtual);
8      double esforcoTratorEfetivoTotal = esforcoTratorEfetivo
9          + composicao.obterComposicao().obterLocomotivaLider().obterEsforcoTratorEfetivoTotal();
10
11     while (esforcoTratorEfetivoTotal > (esforcoTratorAderente * nroLocomotivas)
12             && velocidadeAtual >= 10.0f) {
13         if (pontoAceleracao > 0) {
14             pontoAceleracao--;
15             aplicarPA();
16
17             coeficienteAderencia = composicao.obterComposicao()
18                 .obterSensor().obterCoeficienteDeAderecia();
19             esforcoTratorAderente = composicao.obterComposicao()
20                 .obterEsforcoTratorAderente(coeficienteAderencia, deslocamento.getVelocidadeMedia());
21             esforcoTratorEfetivo = composicao.obterComposicao()
22                 .obterLocomotivaLider().obterEsforcoTrator(potencia, velocidadeAtual);
23             esforcoTratorEfetivoTotal = esforcoTratorEfetivo
24                 + composicao.obterComposicao().obterLocomotivaLider().obterEsforcoTratorEfetivoTotal();
25         }
26     }
27
28     composicao.obterComposicao().obterLocomotivaLider()
29         .setEsforcoTratorEfetivoTotal(esforcoTratorEfetivoTotal);
30
31     return true;
32 }
```

Código 20 - Implementação da função *vaiPatinar*

Nas linhas 2 a 8 é inicializada as variáveis com os valores necessários para os cálculos. Enquanto o esforço trator efetivo total for maior ou igual a 10 e maior que a multiplicação entre o esforço trator aderente e a quantidade de locomotivas que formam a composição, é aplicado um novo ponto de aceleração. Na linha 14, se o PA atual for maior que 0, PA é incrementado e a função *aplicarPA* é novamente executada. Nas linhas 17 a 23, os valores gerados pela função *aplicarPA* necessários para os cálculos são atualizados. Na linha 28, quando aplicado o PA ideal, o valor do esforço trator efetivo total gerado é armazenado na composição.

3.3.4.1Cálculos de condução

Para a realização dos cálculos descritos na seção 2.2, é utilizada a função *calculoMovimento*. O Código 21 ilustra a implementação da função *calculoMovimento*.

```
1  public DeslocamentoDTO calculoMovimento(double velocidadeInicial, double potencia) {
2      this.velocidadeInicial = velocidadeInicial;
3      this.potencia = potencia;
4
5      calcResistenciaLocomotiva();
6      calcResistenciaVagao();
7      calcResistenciaTotal();
8      calcForcaTrator();
9      calcForcaAceleracao();
10     calcVelocidadeFinal();
11     calcDistancia();
12     calcTempo();
13     calcVelocidadeMedia();
14
15     return new DeslocamentoDTO(velocidadeFinal, velocidadeMedia,
16         distanciaPercorrida, tempoGasto, forcaAceleracao, resistenciaTotal);
17 }
```

Código 21 - Implementação da função *calculoMovimento*

Nas linhas 2 e 3 são inicializadas as variáveis globais que armazenam a velocidade atual e potência da composição, respectivamente. Estes valores serão utilizados pelas outras funções para realização dos cálculos.

Na linha 5 é executada a função para cálculo da resistência gerada pelas locomotivas.

```
1  private void calcResistenciaLocomotiva() {
2      resistenciaLocomotiva = Constants.LB_TO_KG.toDouble()
3          * (1.3 + (29 / pesoLocomotiva) + (0.03 * velocidadeInicial)
4          + ((0.024 * areaFrontalLocomotiva * Math.pow(velocidadeInicial, 2))
5              / (pesoLocomotiva * qtdeEixosLocomotiva)));
6  }
```

Código 22 - Implementação da função *calcResistenciaLocomotiva*

Na linha 6 é executada a função para calcular a resistência gerada pelos vagões. O Código 23 ilustra a implementação da função *calcResistenciaVagao*.

```
1  private void calcResistenciaVagao() {
2      resistenciaVagao = Constants.LB_TO_KG.toDouble()
3          * (1.3 + (29 / pesoVagao) + (0.045 * velocidadeInicial)
4          + ((0.0024 * areaFrontalVagao * Math.pow(velocidadeInicial, 2))
5              / (pesoVagao * qtdeEixosVagao)));
6  }
```

Código 23 - Implementação da função *calcResistenciaVagao*

Na linha 7 é executada a função para calcular a resistência total da composição. O Código 24 ilustra a implementação da função *calcResistenciaTotal*.

```
1  private void calcResistenciaTotal() {
2      resistenciaTotal = (pesoLocomotiva * resistenciaLocomotiva)
3          + (pesoVagao * resistenciaVagao);
4  }
```

Código 24 - Implementação da função *calcResistenciaTotal*

Na linha 8 é executada a função para calcular a força trator. O Código 25 ilustra a implementação da função *calcForcaTrator*. No início da execução do agente condutor, por sua velocidade é 0, força trator é setado com valor 10000.

```
1  private void calcForcaTrator() {
2      if (velocidadeInicial == 0) {
3          forcaTrator = 10000;
4      } else {
5          forcaTrator = (273.24 * 0.82 * potencia)
6              / velocidadeInicial;
7      }
8  }
```

Código 25 - Implementação da função *calcForcaTrator*

Na linha 9 é executada a função para calcular a força de aceleração gerada. O Código 26 ilustra a implementação da função *calcForcaAceleracao*.

```
1  private void calcForcaAceleracao() {
2      forcaAceleracao = forcaTrator - resistenciaTotal;
3  }
```

Código 26 - Implementação da função *calcForcaAceleracao*

Na linha 10 é executada a função para calcular a velocidade da composição. O Código 27 ilustra a implementação da função *calcVelocidadeFinal*.

```
1  private void calcVelocidadeFinal() {
2      velocidadeFinal = Math.sqrt(Math.abs(Math.pow(velocidadeInicial, 2)
3          + ((forcaAceleracao * Constants.VARIACAO_DESLOCAMENTO.toInt())
4              / (4 * pesoTotal))));
5  }
```

Código 27 - Implementação da função *calcVelocidadeFinal*

Na linha 11 é executada a função para calcular a distância percorrida pela composição. O Código 28 ilustra a implementação da função *calcDistanciaPercorrida*.

```
1  private void calcDistancia() {
2      distanciaPercorrida = 4 *
3          ((pesoTotal * (Math.pow(velocidadeFinal, 2)
4          - Math.pow(velocidadeInicial, 2))) / forcaAceleracao);
5  }
```

Código 28 - Implementação da função *calcDistanciaPercorrida*

Na linha 12 é executada a função para calcular o tempo gasto para percorrer determinada distância. O Código 29 ilustra a implementação da função *calcTempo*.

```
1  private void calcTempo() {
2      tempoGasto = 7.2 *
3      (distanciaPercorrida / (velocidadeFinal + velocidadeInicial));
4  }
```

Código 29 - Implementação da função *calcTempo*

Na linha 13 é executada a função para calcular a velocidade média da composição. O Código 30 ilustra a implementação da função *calcVelocidadeMedia*.

```
1  private void calcVelocidadeMedia() {
2      velocidadeMedia = (distanciaPercorrida / tempoGasto) * 3.6;
3  }
```

Código 30 - Implementação da função *calcVelocidadeMedia*

Na linha 15, é retornado todos os valores calculados a partir da classe *DeslocamentoDTO*.

Ilustrada pelo Código 31, esta classe foi implementada a fim de facilitar a manipulação destes dados. Os dados armazenados nesta classe são velocidade final, velocidade média, distância percorrida, tempo gasto, força de aceleração e resistência total.

```
1  public class DeslocamentoDTO {
2      private double velocidadeFinal;
3      private double velocidadeMedia;
4      private double distanciaPercorrida;
5      private double tempoGasto;
6      private double forcaAceleracao;
7      private double resistenciaTotal;
8
9      public DeslocamentoDTO() {
10         distanciaPercorrida = 0;
11         tempoGasto = 0;
12     }
13
14     public DeslocamentoDTO(double posicao) {
15         this.distanciaPercorrida = posicao;
16         tempoGasto = 0;
17     }
18
19     public DeslocamentoDTO(double velocidadeFinal, double velocidadeMedia,
20         double distancia, double tempoGasto,
21         double forcaAceleracao, double resistenciaTotal) {
22         this.velocidadeFinal = velocidadeFinal;
23         this.velocidadeMedia = velocidadeMedia;
24         this.distanciaPercorrida = distancia;
25         this.tempoGasto = tempoGasto;
26         this.forcaAceleracao = forcaAceleracao;
27         this.resistenciaTotal = resistenciaTotal;
28     }
29 }
```

Código 31 - Estrutura da classe *DeslocamentoDTO*

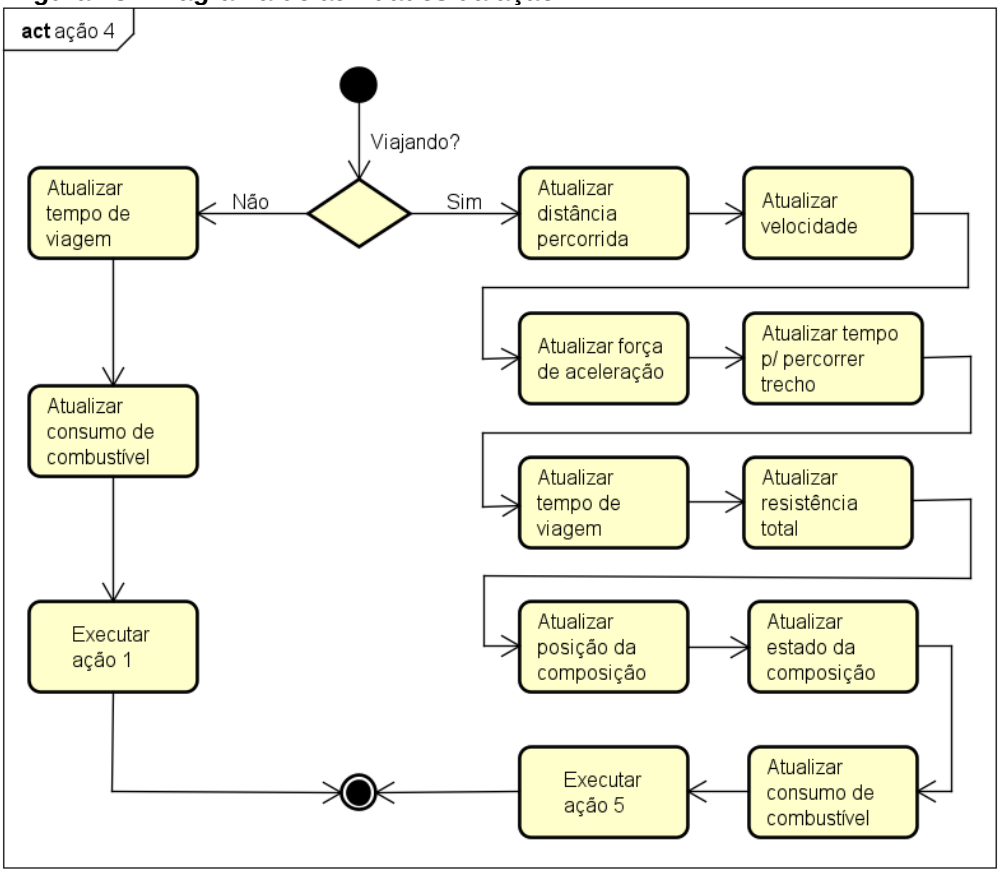
Nas linhas 2 a 7 é instanciada as variáveis para armazenar os valores a serem utilizados. Nas linhas 9 a 27, é implementado os possíveis construtores a serem utilizados da classe.

3.3.5 Ação 4

Em ação 4 são atualizados todos os dados da composição após aplicação e validação do PA na ação 3. A Figura 13 ilustra a ação 4 através de um diagrama de atividades.

Os dados atualizados, caso a composição esteja em viajem, são distância total percorrida, velocidade da composição, força de aceleração gerada, tempo gasto no deslocamento realizado, tempo total de viagem, resistência total da composição, posição atual da composição na via e consumo de combustível.

Figura 13 - Diagrama de atividades da ação 4



Fonte: Autoria própria.

Caso a composição não esteja em viagem, o consumo e o tempo de viagem são atualizados, considerando o tempo de espera até seguir em viagem.

3.3.6 Ação 5

Em ação 5, ilustrada no Código 32, condutor verifica se alcançou seu destino. Caso tenha alcançado seu objetivo, agente condutor envia mensagem ao controlador notificando sobre o fim de sua viagem.

Na linha 2 é verificado se agente condutor alcançou o destino, enviando uma mensagem ao controlador em caso afirmativo. Nas linhas 4 a 14, é enviada uma mensagem a todos os controladores presentes na via sobre o fim de sua viagem. Na linha 15, é incrementado o contador de mensagens enviadas.

```
1   case 5:
2       if (composicao.obterComposicao()
3               .obterHodometro().obterDistanciaPercorrida() >= distanciaViagem) {
4           String str = "finalizar";
5           mensagem = new ACLMessage(ACLMessage.INFORM);
6
7           for (AID c : controladores) {
8               mensagem.addReceiver(c);
9           }
10          mensagem.setConversationId(str);
11          mensagem.setContent(str);
12          mensagem.setReplyWith(myAgent.getLocalName()
13              + System.currentTimeMillis());
14          myAgent.send(mensagem);
15          nroMensagens++;
16
17          viajando = false;
18          acao = -1;
19      } else {
20          acao = 1;
21      }
22      break;
```

Código 32 - Envio de mensagem ao agente controlador

Na linha 17, é configurado como falso a variável que indica que o condutor está em viagem. Na linha 18 a ação do agente é mudada para -1, indicando o fim de sua execução.

Na linha 20, caso a verificação realizada na linha 2 seja falsa, a ação do agente é mudada para 1.

3.4 EXECUÇÃO DOS AGENTES

A execução dos agentes se dá a partir da classe denominada *Main*. O Código 33 ilustra a implementação da classe *Main*.

Nas linhas 4 a 7 é realizado a criação de uma nova instância de JADE e de um novo *Container Principal*, respectivamente. Nas linhas 8 a 10, são carregados os dados da via (cf. seção 3.1.1).

```
1  public class Main {
2      public static void main(String[] args) {
3          try {
4              jade.core.Runtime rt = jade.core.Runtime.instance();
5              Profile p = new ProfileImpl();
6              p.setParameter(Profile.MAIN_HOST, "localhost");
7              ContainerController cc = rt.createMainContainer(p);
8              String caminhoArquivoVia = "\via01.xml";
9              String viagemNro = "01";
10             CIVia via = new CIVia(caminhoArquivoVia, Integer.parseInt(viagemNro));
11             String caminhoArqComposicao1 = "\composicao-lap01.xml";
12             String tipoViagem = "";
13             int nroTentativas = 1;
14             double posicaoEstacao = 0;
15
16             CIComposicao composicao = new CIComposicao(viagemNro,
17                 caminhoArqComposicao1, via, tipoViagem,
18                 nroTentativas, posicaoEstacao);
19             Object[] param = {composicao};
20
21             if (cc != null) {
22                 AgentController agenteControlador = cc.createNewAgent("Estacao_1",
23                     "com.praticDout.agente.AgenteControlador", args);
24                 AgentController agenteCondutor = cc.createNewAgent("Trem_1",
25                     "com.praticDout.agente.AgenteCondutor", param);
26                 agenteControlador.start();
27                 agenteCondutor.start();
28             }
```

Código 33 - Implementação da classe *Main*

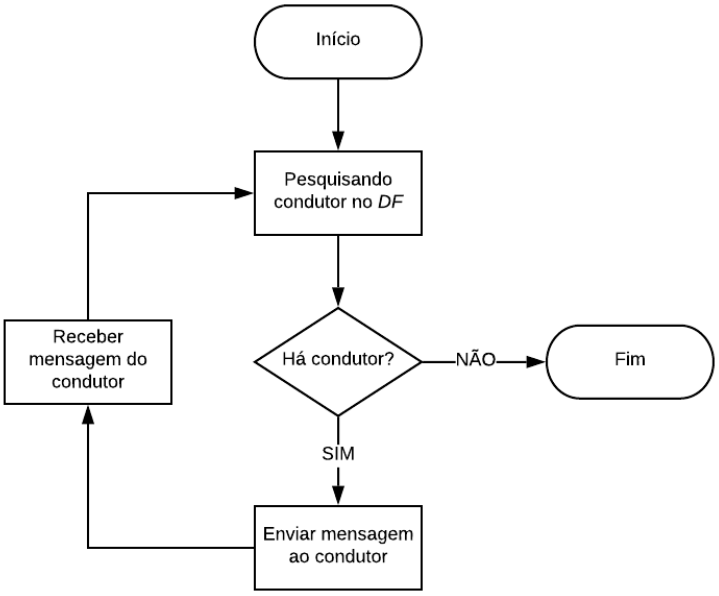
Nas linhas 11 a 16, é carregado os dados da composição (cf. seção 3.1.2). Na linha 19 os dados da composição são armazenados em uma variável do tipo *Object* para posteriormente serem passadas como parâmetro ao agente condutor. Nas linhas 22 e 24, são instanciados agentes do tipo controlador e condutor, respectivamente. Nas linhas 26 e 27, os agentes são inicializados.

3.5 FLUXO DE MENSAGENS

O fluxo de mensagens entre os agentes apresentados anteriormente pode ser representado através de um diagrama, como ilustrado na Figura 14 e na Figura 15.

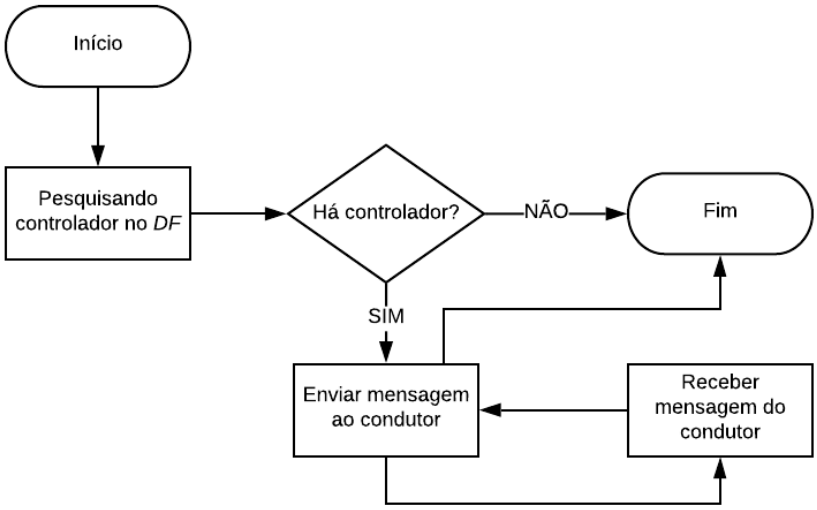
Na Figura 14, enquanto ainda houver agente condutor cadastrado em *DF*, agente controlador troca mensagens com os agentes condutores disponíveis. Quando todos os condutores estiverem descadastrados de *DF*, a execução do agente controlador é finalizada.

Figura 14 - Fluxo de mensagens do agente controlador



Fonte: Autoria própria.

Figura 15 - Fluxo de mensagens do agente condutor



Fonte: Autoria própria.

Na Figura 15, agente condutor busca em *DF* pelo agente controlador. Caso não haja nenhum cadastrado, sua execução é finalizada. Agente condutor troca mensagens com agente controlador até o fim de sua execução.

3.6 CONSIDERAÇÕES FINAIS

Neste capítulo foi apresentado o desenvolvimento do SMA condutor de trens de carga. Foram descritos os arquivos necessários a serem carregados para se obter os dados da composição a ser utilizada nas simulações. Também, foi relatado a implementação dos agentes controlador e condutor, esmiuçando a construção de cada função realizada pelos agentes.

No próximo capítulo será descrito os cenários utilizados para testar o sistema desenvolvido e os resultados obtidos com sua aplicação, expondo valores como velocidade média durante a viagem, consumo médio, consumo total, tempo de viagem e número de mensagens trocadas de cada agente condutor.

4 RESULTADOS

Os agentes controlador e condutor apresentados no capítulo anterior foram implementados com o intuito de apresentar a condução de um trem em uma via a partir da troca de informações com uma estação férrea. Não é o foco deste trabalho a análise aprofundada da condução de um trem de carga, como utilização de desvios e seções de bloqueio, por exemplo.

Na sequência deste capítulo serão apresentados os cenários utilizados para testar o SMA proposto, os resultados obtidos a partir de cada cenário determinado e, por fim, uma análise geral de cada cenário proposto.

4.1 CENÁRIOS DE TESTE

Com o intuito de observar o funcionamento do SMA proposto, a comunicação entre agentes foi analisada em seis cenários diferentes, onde as configurações são apresentadas na Tabela 5. O objetivo destas configurações é analisar o comportamento de trens diferentes sendo conduzidos por agentes, se o controlador é capaz de lidar com um ou mais condutores e se a comunicação entre os agentes obteve sucesso.

Tabela 5 - Configuração dos cenários

Cenário	Número de controladores	Número de condutores	Composição 1	Composição 2
A	1	1	Trem_1	–
B	1	1	Trem_2	–
C	1	1	Trem_3	–
D	1	2	Trem_1	Trem_2
E	1	2	Trem_1	Trem_3
F	1	2	Trem_2	Trem_3

Fonte: Autoria própria.

Para a realização dos experimentos, foram utilizadas três configurações distintas de composição, onde em cada cenário descrito é utilizado alguma dessas composições. A Tabela 6 apresenta as configurações de composição utilizadas. Esta

variação foi utilizada para analisar se os cálculos implementados se os agentes são capazes de conduzir diferentes configurações de trens.

Tabela 6 - Configuração dos agentes condutores

Composição	Quantidade locomotivas	Quantidade vagões	Peso locomotiva (toneladas)	Peso vagão (toneladas)	Peso total composição (toneladas)
Trem_1	4	100	169,7	56,63	6431,8
Trem_2	2	31	169,7	99,56	3425,76
Trem_3	2	28	169,7	99,23	3117,84

Fonte: Autoria própria.

Os cenários descritos na Tabela 5 utilizam as composições descritas na Tabela 6. Os valores utilizados foram selecionados de forma aleatória, visando analisar como a configuração das composições utilizadas influência em valores como tempo de viagem, consumo de combustível e quantidade de mensagens trocadas.

4.2 EXPERIMENTOS

Como descrito na seção 4.1, para cada cenário será utilizado diferentes composições, tendo cada uma dessas diferentes configurações. A seguir serão analisados os cenários descritos na Tabela 5. Os cálculos foram realizados considerando um deslocamento do quilômetro 339 ao quilômetro 330 da via. O número total de mensagens analisadas se refere a quantidade de mensagens trocadas com agente controlador.

Os gráficos apresentados foram gerados a partir da relação velocidade *x* tempo. O fluxo de mensagens dos agentes é apresentado através de trechos de *logs*.

4.2.1 Cenário A

Neste cenário é utilizada apenas uma composição, a composição *Trem_1* da Tabela 6. Os valores obtidos nas simulações são descritos na Tabela 7.

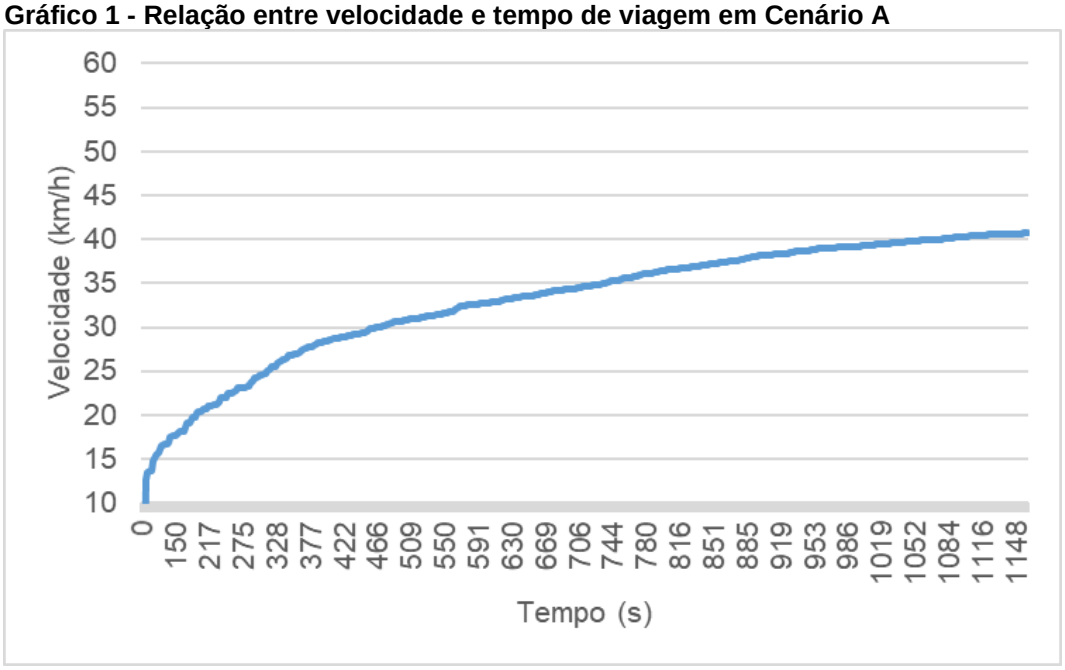
Tabela 7 - Resultados Cenário A

Composição	Velocidade média	Distância percorrida	Consumo total	LTKB	Tempo viagem	Total mensagens
Trem_1	33,02*km/h*	9500*m*	2861,98*l*	0,048*l*	1158,9*s*	475

Fonte: Autoria própria.

O Gráfico 1 apresenta a relação entre velocidade e tempo gasto em um trecho de viagem da composição que forma o Cenário A.

Gráfico 1 - Relação entre velocidade e tempo de viagem em Cenário A



Fonte: Autoria própria.

Por ser o único trem presente na via *Trem_1* pode realizar sua viagem sem preocupações com distância de segurança entre outros trens. O papel do controlador é apenas requisitar que *Trem_1* inicie sua viagem e tomar ciência quando este chegar ao destino final.

O início da troca de mensagens se dá, como ilustrado na Figura 16, pela inicialização dos agentes controlador e condutor. Após *Estacao_1* liberar viagem para *Trem_1* (cf. linha 6), ambos mantêm uma troca de mensagens constante onde o condutor envia valores atualizados sobre sua posição para o controlador.

Figura 16 - *Log* de troca de mensagens entre agentes controlador e condutor

```
1  Controlador: Estacao_1 inicializado!
2  Condutor: Trem_1 inicializado!
3  Estacao_1: pesquisando condutores em DF
4  Trem_1: pesquisando controladores em DF
5  Estacao_1: requisitando posição a todos os condutores
6  Estacao_1: liberando viagem para Trem_1
7  Trem_1: iniciando viagem
8  Trem_1: enviando localização
9  Estacao_1: Trem_1 enviou sua posição
10 Estacao_1: pesquisando condutores em DF
11 Estacao_1: requisitando posição a todos os condutores
12 Trem_1: enviando localização
13 Estacao_1: Trem_1 enviou sua posição
```

Fonte: Autoria própria.

Ao final de sua viagem, como ilustra a Figura 17, *Trem_1* informa a *Estacao_1* sobre o fim de sua viagem (cf. linha 4) e termina sua execução (cf. linha 5), finalizando o ciclo de mensagens entre ambos.

Figura 17 - *Log* de finalização de *Trem_1*

```
1  Estacao_1: pesquisando condutores em DF
2  Estacao_1: requisitando posição a todos os condutores
3  Trem_1: enviando localização
4  Trem_1: informando fim da viagem
5  Condutor: Trem_1 finalizado!
6  Estacao_1: Trem_1 enviou sua posição
7  Estacao_1: Trem_1 chegou ao destino
8  Estacao_1: pesquisando condutores em DF
9  Controlador: Estacao_1 finalizado!
```

Fonte: Autoria própria.

Ao receber de *DF* que não há mais nenhum condutor na via, a execução de *Estacao_1* também é finalizada (cf. linha 9).

4.2.2 Cenário B

Neste cenário é utilizado apenas uma composição, a composição *Trem_2* da Tabela 6. Os valores obtidos nas simulações são descritos na Tabela 8.

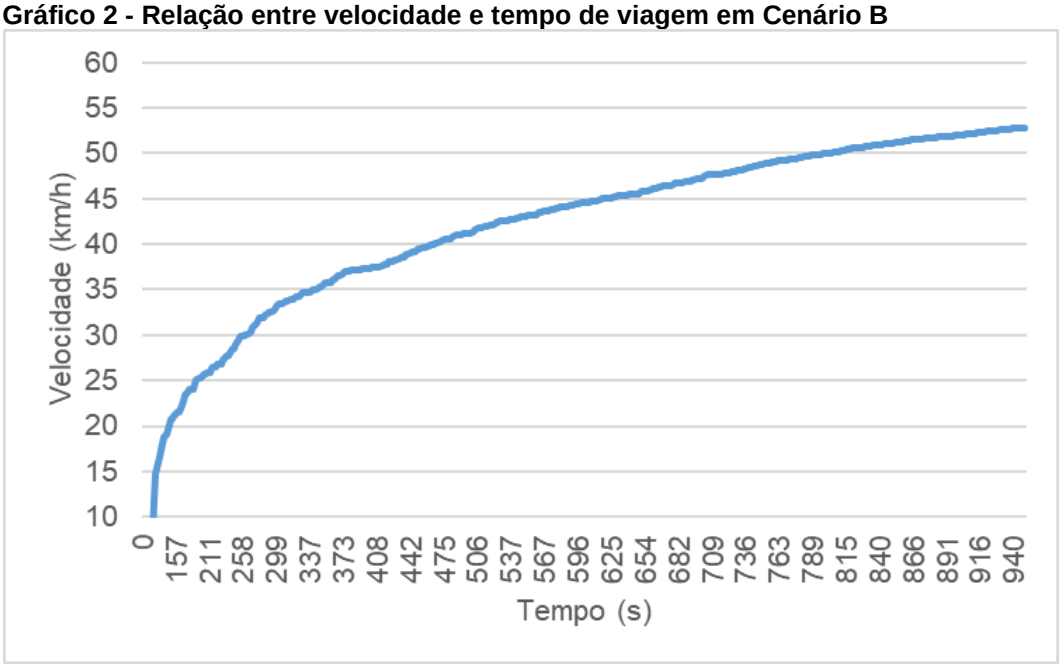
Tabela 8 - Resultados Cenário B

Composição	Velocidade média	Distância percorrida	Consumo total	LTKB	Tempo viagem	Total mensagens
Trem_2	42,07*km/h*	9500*m*	2736,78*l*	0,084*l*	948,57*s*	475

Fonte: Autoria própria.

O Gráfico 2 apresenta a relação entre velocidade e tempo gasto em um trecho de viagem da composição que forma o Cenário B.

Gráfico 2 - Relação entre velocidade e tempo de viagem em Cenário B



Fonte: Autoria própria.

Por ser o único trem presente na via *Trem_2* pode realizar sua viagem sem preocupações com distância de segurança entre outros trens. O papel do controlador é apenas requisitar que *Trem_2* inicie sua viagem e tomar ciência quando este chegar ao destino final.

É interessante observar a diferença de valores obtidos entre os cenários A e B. Embora os dois trens estejam viajando sozinhos em seus respectivos cenários, *Trem_2* é mais leve em comparação a *Trem_1*, conseguindo assim resultados melhores em relação a velocidade média, tempo total de viagem e consumo total de combustível.

O início da troca de mensagens se dá, como ilustrado na Figura 18, pela inicialização dos agentes controlador e condutor. Após *Estacao_1* liberar viagem para

Trem_2 (cf. linha 6), ambos mantêm uma troca de mensagens constante onde o condutor envia valores atualizados sobre sua posição para o controlador.

Figura 18 - *Log* de troca de mensagens entre agentes controlador e condutor

```
1  Controlador: Estacao_1 inicializado!
2  Condutor: Trem_2 inicializado!
3  Estacao_1: pesquisando condutores em DF
4  Trem_2: pesquisando controladores em DF
5  Estacao_1: requisitando posição a todos os condutores
6  Estacao_1: liberando viagem para Trem_2
7  Trem_2: iniciando viagem
8  Trem_2: enviando localização
9  Estacao_1: Trem_2 enviou sua posição
10 Estacao_1: pesquisando condutores em DF
11 Estacao_1: requisitando posição a todos os condutores
12 Trem_2: enviando localização
13 Estacao_1: Trem_2 enviou sua posição
```

Fonte: Autoria própria.

Ao final de sua viagem, como ilustra a Figura 19, *Trem_2* informa a *Estacao_1* sobre o fim de sua viagem (cf. linha 4) e termina sua execução (cf. linha 5), finalizando o ciclo de mensagens entre ambos.

Figura 19 - *Log* de finalização de *Trem_2*

```
1  Estacao_1: pesquisando condutores em DF
2  Estacao_1: requisitando posição a todos os condutores
3  Trem_2: enviando localização
4  Trem_2: informando fim da viagem
5  Condutor: Trem_2 finalizado!
6  Estacao_1: Trem_2 enviou sua posição
7  Estacao_1: Trem_2 chegou ao destino
8  Estacao_1: pesquisando condutores em DF
9  Controlador: Estacao_1 finalizado!
```

Fonte: Autoria própria.

Ao receber de *DF* que não há mais nenhum condutor na via, a execução de *Estacao_1* também é finalizada (cf. linha 9).

4.2.3 Cenário C

Neste cenário é utilizado apenas uma composição, a composição *Trem_3* da Tabela 6. Os valores obtidos nas simulações são descritos na Tabela 9.

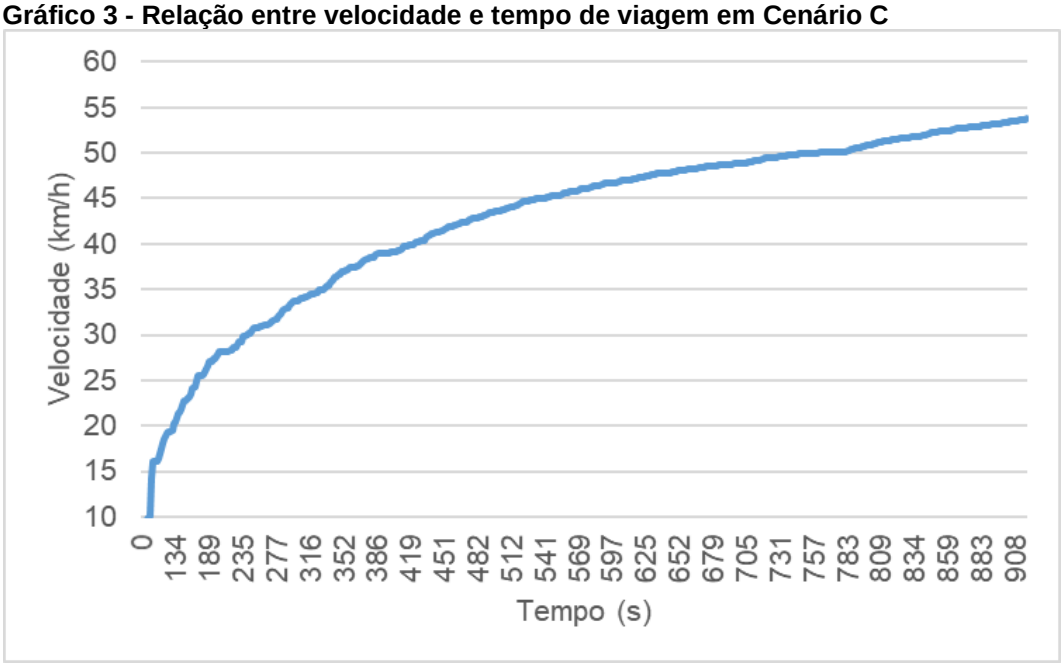
Tabela 9 - Resultados Cenário C

Composição	Velocidade média	Distância percorrida	Consumo total	LTKB	Tempo viagem	Total mensagens
Trem_3	42,74*km/h*	9500*m*	2592,68*l*	0,088*l*	915,79*s*	475

Fonte: Autoria própria.

O Gráfico 3 apresenta a relação entre velocidade e tempo gasto em um trecho de viagem da composição que forma o Cenário C.

Gráfico 3 - Relação entre velocidade e tempo de viagem em Cenário C



Fonte: Autoria própria.

Por ser o único trem presente na via *Trem_3* pode realizar sua viagem sem preocupações com distância de segurança entre outros trens. O papel do controlador é apenas requisitar que *Trem_3* inicia sua viagem e tomar ciência quando este chegar ao destino final.

É interessante observar a diferença de valores obtidos entre os cenários B e C. A diferença de peso total e quantidade de vagões utilizados nas composições *Trem_2* e *Trem_3* são pequenas, tornando *Trem_3* um pouco mais leve em relação a

Trem_2, conseguindo assim valores um pouco melhores em relação a velocidade média e tempo total de viagem.

O início da troca de mensagens se dá, como ilustrado na Figura 20, pela inicialização dos agentes controlador e condutor. Após *Estacao_1* liberar viagem para *Trem_3* (cf. linha 6), ambos mantêm uma troca de mensagens constante onde o condutor envia valores atualizados sobre sua posição para o controlador.

Figura 20 - *Log* de troca de mensagens entre agentes controlador e condutor

```
1  Controlador: Estacao_1 inicializado!
2  Condutor: Trem_3 inicializado!
3  Estacao_1: pesquisando condutores em DF
4  Trem_3: pesquisando controladores em DF
5  Estacao_1: requisitando posição a todos os condutores
6  Estacao_1: liberando viagem para Trem_3
7  Trem_3: iniciando viagem
8  Trem_3: enviando localização
9  Estacao_1: Trem_3 enviou sua posição
10 Estacao_1: pesquisando condutores em DF
11 Estacao_1: requisitando posição a todos os condutores
12 Trem_3: enviando localização
13 Estacao_1: Trem_3 enviou sua posição
```

Fonte: Autoria própria.

Ao final de sua viagem, como ilustra a Figura 21, *Trem_3* informa a *Estacao_1* sobre o fim de sua viagem (cf. linha 4) e termina sua execução (cf. linha 5), finalizando o ciclo de mensagens entre ambos.

Figura 21 - *Log* de finalização de *Trem_3*

```
1  Estacao_1: pesquisando condutores em DF
2  Estacao_1: requisitando posição a todos os condutores
3  Trem_3: enviando localização
4  Trem_3: informando fim da viagem
5  Condutor: Trem_3 finalizado!
6  Estacao_1: Trem_3 enviou sua posição
7  Estacao_1: Trem_3 chegou ao destino
8  Estacao_1: pesquisando condutores em DF
9  Controlador: Estacao_1 finalizado!
```

Fonte: Autoria própria.

Ao receber de *DF* que não há mais nenhum condutor na via, a execução de *Estacao_1* também é finalizada (cf. linha 9).

4.2.4 Cenário D

Neste cenário há a presença de dois trens na via, compreendido pelas composições *Trem_1* e *Trem_2*. Os valores obtidos nas simulações são descritos na Tabela 10.

Tabela 10 - Resultados Cenário D

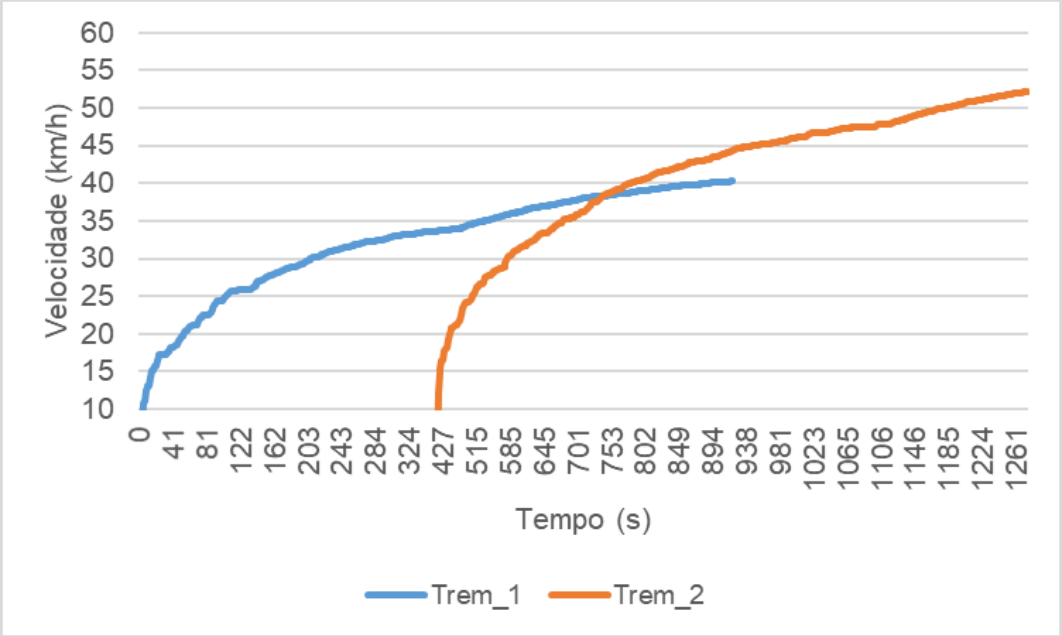
Composição	Velocidade média	Distância percorrida	Consumo total	LTKB	Tempo viagem	Total mensagens
Trem_1	32,23*km/h*	9500*m*	2779,26*l*	0,046*l*	1200,84*s*	475
Trem_2	41,48*km/h*	9500*m*	3056,31*l*	0,094*l*	1273,72*s*	712
		Total			**2474,56**	**1187**

Fonte: Autoria própria.

O Gráfico 4 apresenta a relação entre velocidade e tempo gasto em um trecho de viagem das composições que formam o Cenário D.

Neste cenário, é possível notar a diferença de total de mensagens enviadas ao controlador e tempo de viagem entre as composições. Para respeitar a distância de segurança estabelecida, *Trem_2* só segue viagem quando *Trem_1* ultrapassar a distância de segurança estabelecida. Enquanto está em espera, *Trem_2* mantém troca de mensagens constante com o controlador, até que sua viagem seja liberada.

Gráfico 4 - Relação entre velocidade e tempo de viagem em Cenário D



Fonte: Autoria própria.

O início da troca de mensagens se dá, como ilustrado na Figura 22, pela inicialização dos agentes controlador e condutor. Após *Estacao_1* liberar viagem para *Trem_1* (cf. linha 8), ambos os condutores mantêm uma troca de mensagens constante com o controlador, onde os condutores enviam valores atualizados sobre suas respectivas posições.

Figura 22 - *Log* de troca de mensagens entre agentes controlador e condutores

```
1  Controlador: Estacao_1 inicializado!
2  Condutor: Trem_1 inicializado!
3  Condutor: Trem_2 inicializado!
4  Trem_1: pesquisando controladores em DF
5  Trem_2: pesquisando controladores em DF
6  Estacao_1: pesquisando condutores em DF
7  Estacao_1: requisitando posição a todos os condutores
8  Estacao_1: liberando viagem para Trem_1
9  Trem_1: iniciando viagem
10 Trem_1: enviando localização
11 Trem_2: enviando localização
12 Estacao_1: Trem_1 enviou sua posição
13 Estacao_1: Trem_2 enviou sua posição
14 Estacao_1: pesquisando condutores em DF
15 Estacao_1: requisitando posição a todos os condutores
16 Trem_1: enviando localização
17 Trem_2: enviando localização
18 Estacao_1: Trem_1 enviou sua posição
19 Estacao_1: Trem_2 enviou sua posição
```

Fonte: Autoria própria.

Quando liberado, *Estacao_1* envia permissão a *Trem_2* (cf. linha 7) para o início de sua viagem, conforme ilustra a Figura 23. Por sua vez, *Trem_2* responde a requisição para indicar que iniciou sua viagem (cf. linha 12).

Figura 23 - *Log* de *trem_2* iniciando viagem

```
1  Trem_1: enviando localização
2  Trem_2: enviando localização
3  Estacao_1: Trem_1 enviou sua posição
4  Estacao_1: Trem_2 enviou sua posição
5  Estacao_1: pesquisando condutores em DF
6  Estacao_1: requisitando posição a todos os condutores
7  Estacao_1: liberando viagem para Trem_2
8  Trem_1: enviando localização
9  Trem_2: iniciando viagem
10 Trem_2: enviando localização
11 Estacao_1: Trem_1 enviou sua posição
12 Estacao_1: Trem_2 seguindo em viagem
13 Estacao_1: Trem_2 enviou sua posição
14 Estacao_1: pesquisando condutores em DF
15 Estacao_1: requisitando posição a todos os condutores
16 Trem_1: enviando localização
17 Trem_2: enviando localização
```

Fonte: Autoria própria.

Ao alcançar o fim da viagem, como ilustra a Figura 24, *Trem_1* informa *Estacao_1* por meio de uma mensagem que alcançou seu objetivo (cf. linha 10).

Figura 24 - *Log* de finalização de *Trem_1*

```
1  Estacao_1: pesquisando condutores em DF
2  Estacao_1: requisitando posição a todos os condutores
3  Trem_1: enviando localização
4  Trem_2: enviando localização
5  Estacao_1: Trem_1 enviou sua posição
6  Estacao_1: Trem_2 enviou sua posição
7  Estacao_1: pesquisando condutores em DF
8  Estacao_1: requisitando posição a todos os condutores
9  Trem_1: enviando localização
10 Trem_1: informando fim da viagem
11 Trem_2: enviando localização
12 Condutor: Trem_1 finalizado!
13 Estacao_1: Trem_1 enviou sua posição
14 Estacao_1: Trem_1 chegou ao destino
15 Estacao_1: Trem_2 enviou sua posição
16 Estacao_1: pesquisando condutores em DF
17 Estacao_1: requisitando posição a todos os condutores
```

Fonte: Autoria própria.

Ao final de sua viagem, como ilustra a Figura 25, *Trem_2* informa a *Estacao_1* sobre o fim de sua viagem (cf. linha 4) e termina sua execução (cf. linha 7).

Figura 25 - *Log* de finalização de *Trem_2*

```
1  Estacao_1: pesquisando condutores em DF
2  Estacao_1: requisitando posição a todos os condutores
3  Trem_2: enviando localização
4  Trem_2: informando fim da viagem
5  Estacao_1: Trem_2 enviou sua posição
6  Estacao_1: Trem_2 chegou ao destino
7  Condutor: Trem_2 finalizado!
8  Estacao_1: pesquisando condutores em DF
9  Controlador: Estacao_1 finalizado!
```

Fonte: Autoria própria.

Ao receber de *DF* que não há mais nenhum condutor na via, a execução de *Estacao_1* também é finalizada (cf. linha 9).

4.2.5 Cenário E

Neste cenário há a presença de dois trens na via, compreendido pelas composições *Trem_1* e *Trem_3*. Os valores obtidos nas simulações são descritos na Tabela 11.

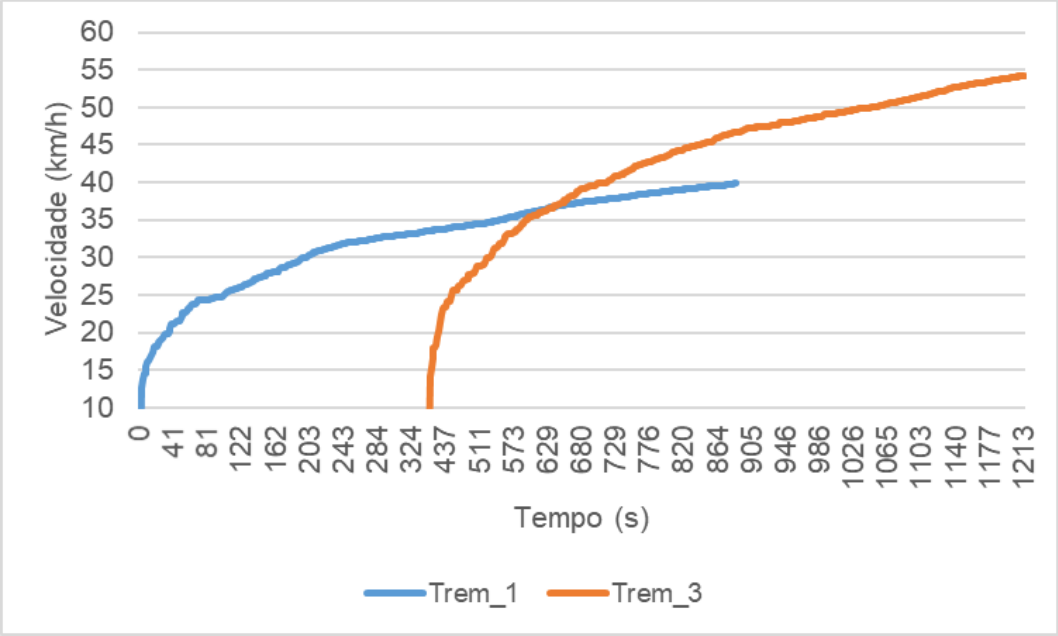
Tabela 11 - Resultados Cenário E

Composição	Velocidade média	Distância percorrida	Consumo total	LTKB	Tempo viagem	Total mensagens
Trem_1	32,39*km/h*	9500*m*	2744,87*l*	0,046*l*	1167,43*s*	475
Trem_3	43,45*km/h*	9500*m*	3056,76*l*	0,103*l*	1214,23*s*	704
		Total			**2381,66**	**1179**

Fonte: Autoria própria.

O Gráfico 5 apresenta a relação entre velocidade e tempo gasto em um trecho de viagem das composições que formam o Cenário E.

Gráfico 5 - Relação entre velocidade e tempo de viagem em Cenário E



Fonte: Autoria própria.

Como no cenário anterior, é possível notar a diferença de total de mensagens enviadas ao controlador e tempo de viagem entre as composições. Para respeitar a distância de segurança estabelecida, *Trem_3* só segue com sua viagem quando *Trem_1* ultrapassar a distância de segurança estabelecida. Enquanto está em espera, *Trem_3* mantém troca de mensagens constante com o controlador, até que sua viagem seja liberada.

O início da troca de mensagens se dá, como ilustrado na Figura 26, pela inicialização dos agentes controlador e condutor. Após *Estacao_1* liberar viagem para *Trem_1* (cf. linha 8), ambos os condutores mantêm uma troca de mensagens constante com o controlador, onde os condutores enviam valores atualizados sobre suas respectivas posições.

Figura 26 - *Log* de troca de mensagens entre agentes controlador e condutores

```
1  Controlador: Estacao_1 inicializado!
2  Condutor: Trem_1 inicializado!
3  Condutor: Trem_3 inicializado!
4  Trem_1: pesquisando controladores em DF
5  Trem_3: pesquisando controladores em DF
6  Estacao_1: pesquisando condutores em DF
7  Estacao_1: requisitando posição a todos os condutores
8  Estacao_1: liberando viagem para Trem_1
9  Trem_1: iniciando viagem
10 Trem_1: enviando localização
11 Trem_3: enviando localização
12 Estacao_1: Trem_1 enviou sua posição
13 Estacao_1: Trem_3 enviou sua posição
14 Estacao_1: pesquisando condutores em DF
15 Estacao_1: requisitando posição a todos os condutores
16 Trem_1: enviando localização
17 Trem_3: enviando localização
18 Estacao_1: Trem_1 enviou sua posição
19 Estacao_1: Trem_3 enviou sua posição
```

Fonte: Autoria própria.

Quando liberado, *Estacao_1* envia permissão a *Trem_3* (cf. linha 7) para o início de sua viagem, conforme ilustra a Figura 27. Por sua vez, *Trem_3* responde a requisição para indicar que iniciou sua viagem (cf. linha 12).

Figura 27 - *Log* de *Trem_3* iniciando viagem

```
1  Trem_1: enviando localização
2  Trem_3: enviando localização
3  Estacao_1: Trem_1 enviou sua posição
4  Estacao_1: Trem_3 enviou sua posição
5  Estacao_1: pesquisando condutores em DF
6  Estacao_1: requisitando posição a todos os condutores
7  Estacao_1: liberando viagem para Trem_3
8  Trem_1: enviando localização
9  Trem_3: iniciando viagem
10 Trem_3: enviando localização
11 Estacao_1: Trem_1 enviou sua posição
12 Estacao_1: Trem_3 seguindo em viagem
13 Estacao_1: Trem_3 enviou sua posição
14 Estacao_1: pesquisando condutores em DF
15 Estacao_1: requisitando posição a todos os condutores
16 Trem_1: enviando localização
17 Trem_3: enviando localização
```

Fonte: Autoria própria.

Ao alcançar o fim da viagem, como ilustra a Figura 28, *Trem_1* informa *Estacao_1* por meio de uma mensagem que alcançou seu objetivo (cf. linha 10).

Figura 28 - *Log* de finalização de *Trem_1*

```
1  Estacao_1: pesquisando condutores em DF
2  Estacao_1: requisitando posição a todos os condutores
3  Trem_1: enviando localização
4  Trem_3: enviando localização
5  Estacao_1: Trem_1 enviou sua posição
6  Estacao_1: Trem_3 enviou sua posição
7  Estacao_1: pesquisando condutores em DF
8  Estacao_1: requisitando posição a todos os condutores
9  Trem_1: enviando localização
10 Trem_1: informando fim da viagem
11 Trem_3: enviando localização
12 Condutor: Trem_1 finalizado!
13 Estacao_1: Trem_1 enviou sua posição
14 Estacao_1: Trem_1 chegou ao destino
15 Estacao_1: Trem_3 enviou sua posição
16 Estacao_1: pesquisando condutores em DF
17 Estacao_1: requisitando posição a todos os condutores
```

Fonte: Autoria própria.

Ao final de sua viagem, como ilustra a Figura 29, *Trem_3* informa a *Estacao_1* sobre o fim de sua viagem (cf. linha 4) e termina sua execução (cf. linha 7), finalizando o ciclo de mensagens entre ambos.

Figura 29 - *Log* de finalização de *Trem_3*

```
1  Estacao_1: pesquisando condutores em DF
2  Estacao_1: requisitando posição a todos os condutores
3  Trem_3: enviando localização
4  Trem_3: informando fim da viagem
5  Estacao_1: Trem_3 enviou sua posição
6  Estacao_1: Trem_3 chegou ao destino
7  Condutor: Trem_3 finalizado!
8  Estacao_1: pesquisando condutores em DF
9  Controlador: Estacao_1 finalizado!
```

Fonte: Autoria própria.

Ao receber de *DF* que não há mais nenhum condutor na via, a execução de *Estacao_1* também é finalizada (cf. linha 9).

4.2.6 Cenário F

Neste cenário há a presença de dois trens na via, compreendido pelas composições *Trem_2* e *Trem_3*. Os valores obtidos nas simulações são descritos na Tabela 12.

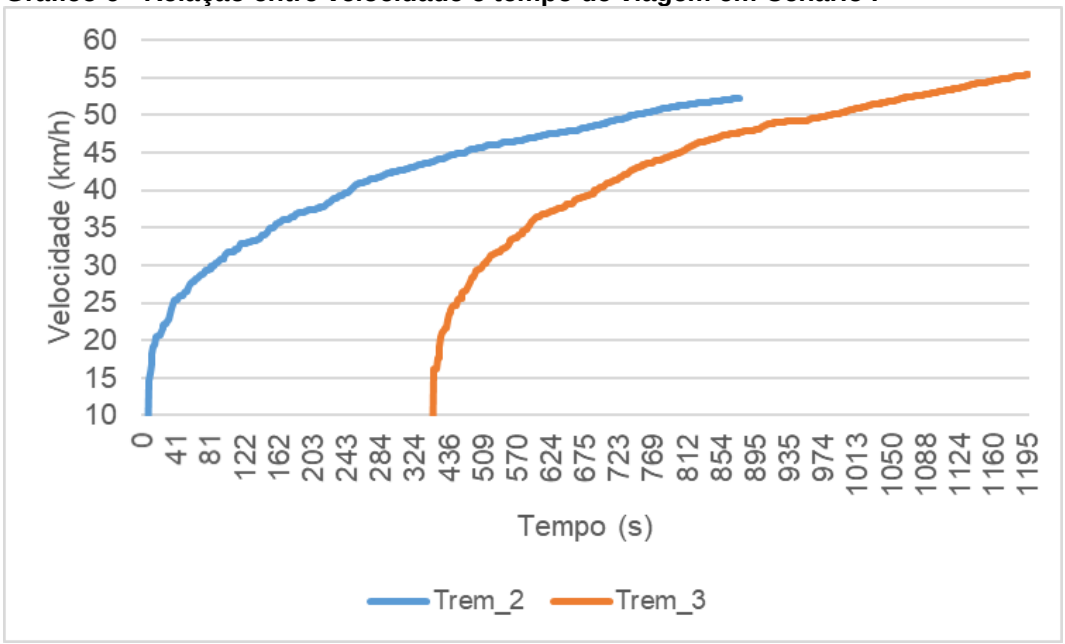
Tabela 12 - Resultados Cenário F

Composição	Velocidade média	Distância percorrida	Consumo total	LTKB	Tempo viagem	Total mensagens
Trem_2	41,57*km/h*	9500*m*	2658,15*l*	0,082*l*	949,45*s*	475
Trem_3	44,42*km/h*	9500*m*	3187,12*l*	0,108*l*	1196,66*s*	704
		Total			**2146,11**	**1179**

Fonte: Autoria própria.

O Gráfico 6 apresenta a relação entre velocidade e tempo gasto em um trecho de viagem das composições que formam o Cenário F.

Gráfico 6 - Relação entre velocidade e tempo de viagem em Cenário F



Fonte: Autoria própria.

Neste cenário, é possível observar que as composições *Trem_2* e *Trem_3* possuem valores finais muito próximos pelo fato de as configurações das composições utilizadas serem parecidas.

É possível visualizar que *Trem_3* permanece em sua posição inicial até o momento que *Trem_2* atinge a distância de segurança. Quando alcançada, agente controlador envia uma mensagem a *Trem_3* permitindo sua viagem.

O início da troca de mensagens se dá, como ilustrado na Figura 30, pela inicialização dos agentes controlador e condutor. Após *Estacao_1* liberar viagem para *Trem_2* (cf. linha 8), ambos os condutores mantêm uma troca de mensagens

constante com o controlador, onde os condutores enviam valores atualizados sobre suas respectivas posições.

Figura 30 - *Log* de troca de mensagens entre agentes controlador e condutores

```
1  Controlador: Estacao_1 inicializado!
2  Condutor: Trem_2 inicializado!
3  Condutor: Trem_3 inicializado!
4  Trem_2: pesquisando controladores em DF
5  Trem_3: pesquisando controladores em DF
6  Estacao_1: pesquisando condutores em DF
7  Estacao_1: requisitando posição a todos os condutores
8  Estacao_1: liberando viagem para Trem_2
9  Trem_2: iniciando viagem
10 Trem_2: enviando localização
11 Trem_3: enviando localização
12 Estacao_1: Trem_2 enviou sua posição
13 Estacao_1: Trem_3 enviou sua posição
14 Estacao_1: pesquisando condutores em DF
15 Estacao_1: requisitando posição a todos os condutores
16 Trem_2: enviando localização
17 Trem_3: enviando localização
18 Estacao_1: Trem_2 enviou sua posição
19 Estacao_1: Trem_3 enviou sua posição
```

Fonte: Autoria própria.

Quando liberado, *Estacao_1* envia permissão a *Trem_3* (cf. linha 7) para o início de sua viagem, conforme ilustra a Figura 31. Por sua vez, *Trem_3* responde a requisição para indicar que iniciou sua viagem (cf. linha 12).

Figura 31 - *Log* de *Trem_3* iniciando viagem

```
1  Trem_2: enviando localização
2  Trem_3: enviando localização
3  Estacao_1: Trem_2 enviou sua posição
4  Estacao_1: Trem_3 enviou sua posição
5  Estacao_1: pesquisando condutores em DF
6  Estacao_1: requisitando posição a todos os condutores
7  Estacao_1: liberando viagem para Trem_3
8  Trem_2: enviando localização
9  Trem_3: iniciando viagem
10 Trem_3: enviando localização
11 Estacao_1: Trem_2 enviou sua posição
12 Estacao_1: Trem_3 seguindo em viagem
13 Estacao_1: Trem_3 enviou sua posição
14 Estacao_1: pesquisando condutores em DF
15 Estacao_1: requisitando posição a todos os condutores
16 Trem_2: enviando localização
17 Trem_3: enviando localização
```

Fonte: Autoria própria.

Ao alcançar o fim da viagem, como ilustra a Figura 32, *Trem_2* informa *Estacao_1* por meio de uma mensagem que alcançou seu objetivo (cf. linha 10).

Figura 32 - *Log* de finalização de *Trem_2*

```
1  Estacao_1: pesquisando condutores em DF
2  Estacao_1: requisitando posição a todos os condutores
3  Trem_2: enviando localização
4  Trem_3: enviando localização
5  Estacao_1: Trem_2 enviou sua posição
6  Estacao_1: Trem_3 enviou sua posição
7  Estacao_1: pesquisando condutores em DF
8  Estacao_1: requisitando posição a todos os condutores
9  Trem_2: enviando localização
10 Trem_2: informando fim da viagem
11 Trem_3: enviando localização
12 Condutor: Trem_2 finalizado!
13 Estacao_1: Trem_2 enviou sua posição
14 Estacao_1: Trem_2 chegou ao destino
15 Estacao_1: Trem_3 enviou sua posição
16 Estacao_1: pesquisando condutores em DF
17 Estacao_1: requisitando posição a todos os condutores
```

Fonte: Autoria própria.

Ao final de sua viagem, como ilustra a Figura 33, *Trem_3* informa a *Estacao_1* sobre o fim de sua viagem (cf. linha 4) e termina sua execução (cf. linha 7), finalizando o ciclo de mensagens entre ambos.

Figura 33 - *Log* de finalização de *Trem_3*

```
1  Estacao_1: pesquisando condutores em DF
2  Estacao_1: requisitando posição a todos os condutores
3  Trem_3: enviando localização
4  Trem_3: informando fim da viagem
5  Estacao_1: Trem_3 enviou sua posição
6  Estacao_1: Trem_3 chegou ao destino
7  Condutor: Trem_3 finalizado!
8  Estacao_1: pesquisando condutores em DF
9  Controlador: Estacao_1 finalizado!
```

Fonte: Autoria própria.

Ao receber de *DF* que não há mais nenhum condutor na via, a execução de *Estacao_1* também é finalizada (cf. linha 9).

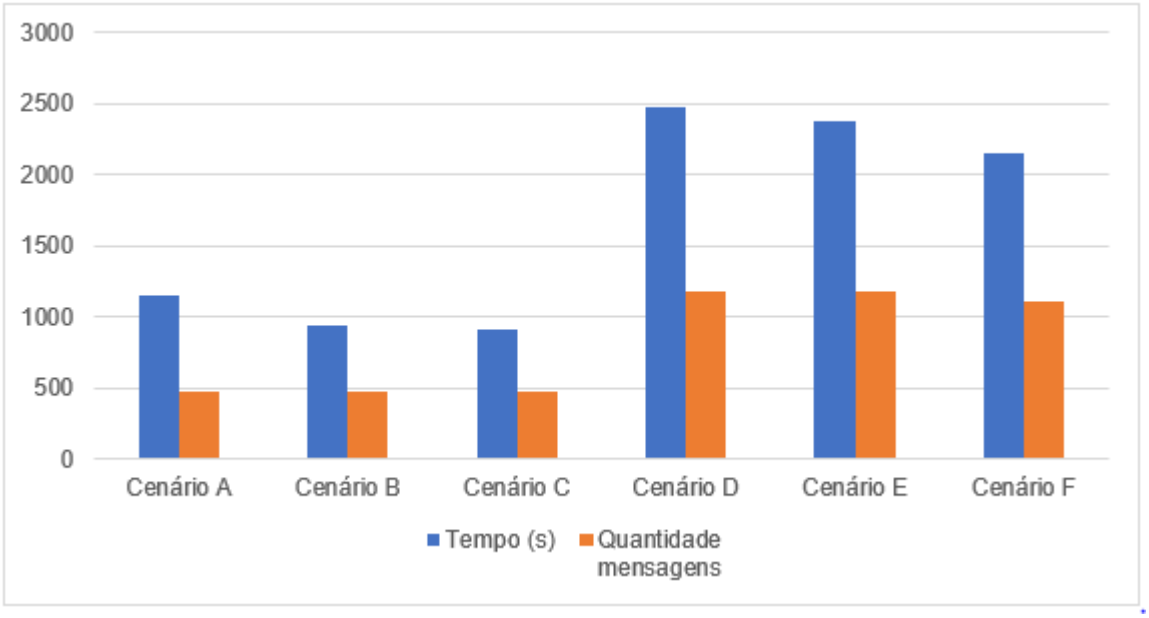
4.3 COMPARATIVO ENTRE CENÁRIOS

Com os resultados obtidos em cada cenário descrito, é possível descrever comparativos entre os cenários abordados.

Em todos os cenários é possível verificar oscilações na velocidade das composições. Esta oscilação ocorre devido a escolha dos pontos de aceleração se darem de forma aleatória, gerando potências que podem diminuir a velocidade atual da locomotiva.

O Gráfico 7 apresenta uma comparação entre os resultados obtidos em cada cenário em relação a tempo total de viagem – em segundos – e troca de mensagens pelo agente controlador.

Gráfico 7 - Comparativo entre os cenários estudados



Fonte: Autoria própria.

A diferença de tempo de viagem entre os cenários A, B e C se dá pela configuração das composições que formam cada um destes. Por ter seu peso total mais leve, *Trem_3* (cenário C) consegue alcançar velocidades maiores em relação a *Trem_2* (cenário B) e principalmente *Trem_1* (cenário A), a composição mais pesada entre as três.

A relação de tempo de viagem entre os cenários D, E e F também mantém determinado padrão por conta das composições que formam tais cenários. Cenário D

apresenta um maior tempo de viagem pelo fato de *Trem_1*, por alcançar velocidades menores por causa de seu peso, demorar determinado período para alcançar a distância necessária e assim *Trem_2* iniciar sua viagem. Em contrapartida cenário C possui o menor tempo de viagem entre os cenários citados pelo fato de ser composto pelas composições *Trem_2* e *Trem_3*, ambas com configurações semelhantes, alcançando assim velocidades maiores comparadas a *Trem_1*.

É importante destacar também a efetiva troca de mensagens entre os diversos condutores com o controlador. Fica evidente o fluxo de mensagens entre estes agentes em cada um dos cenários testados, permitindo que o controlador mantenha o gerenciamento dos condutores presentes na via através do envio constante de suas respectivas posições atuais.

A quantidade de mensagens entre condutor e controlador se manteve a mesma entre os cenários constituídos por apenas uma composição. Nos cenários constituídos por duas composições fica evidente o aumento do número de mensagens enviadas ao controlador por parte da segunda composição em relação a primeira composição do cenário. Esta diferença de total de mensagens está relacionada ao tempo de espera por parte da segunda composição até que seja liberada sua viagem, mantendo uma troca de mensagens constante com o controlador enquanto a primeira composição não ultrapassa a distância mínima de segurança.

4.4 CONSIDERAÇÕES FINAIS

Este capítulo apresentou os resultados encontrados com a utilização do sistema desenvolvido, a partir da utilização de seis cenários. Com a aplicação destes, ficou evidente a diferença entre cada configuração de composição, além de valores gerados como velocidade média de viagem, tempo total de viagem e mensagens trocadas entre controlador e condutores. Com estes resultados é possível desenvolver as conclusões finais, descritas no capítulo seguinte.

5 CONCLUSÃO

O modal férreo é um dos meios mais viáveis para transportes de carga, possuindo grande espaço para redução de custos, o que incentiva pesquisas e inovações neste setor. A utilização de recursos tecnológicos que auxiliem na redução de, por exemplo, consumo de combustível durante uma viagem, pode acarretar na diminuição dos custos anuais de operação.

O simulador de condução de trens de carga proposto por Borges (2015) tem como intuito o aprimoramento da técnica de condução assistida de trens de carga a partir de geração de políticas ou planos de ações para se conduzir um trem de carga em determinada via férrea. Tomando este simulador como base, foi desenvolvido a implementação de um sistema multiagente com a capacidade de simular a condução de trens de carga em dada via férrea.

O escopo do trabalho apresentado contempla a troca de mensagens por parte das diversas composições presentes na via com a estação férrea, permitindo a essa o controle e organização da via. Também, é contemplado a condução das composições pela via, excluindo do escopo do trabalho a busca por otimizações na condução.

Utilizando *framework* JADE, foram implementados agentes do tipo controlador e condutor, com trocas de mensagens entre ambos para obtenção de informações do agente condutor por parte do agente controlador e a realização dos cálculos necessários para a condução por parte do agente condutor. A utilização de JADE se dá principalmente por seu principal objetivo de facilitar o desenvolvimento de sistemas baseados em agentes.

Para os resultados foram apresentados alguns cenários, com cada cenário contendo diferentes configurações de composição, como número de locomotivas e número de vagões que as compõem. Com estes cenários ficaram evidentes algumas características, como a diferença de velocidade média e tempo total de viagem de um trem de acordo com suas configurações, além da quantidade de mensagens trocadas entre o agente condutor e agente controlador.

5.1 TRABALHOS FUTUROS

Como trabalhos futuros espera-se a aplicação de controle de seções de bloqueio e seções de desvio na via. Outro ponto a ser trabalhado é a otimização da condução dos trens utilizando o SMA desenvolvido, buscando configurações de via e tomada de ações pelo agente controlador que acarrete uma viagem no menor tempo possível, com menor consumo de combustível possível.

Pode ser aplicado a este trabalho, técnicas de aprendizagem aos agentes visando o aprimoramento do SMA desenvolvido e a otimização da condução. Também, é possível realizar a interligação deste trabalho com o computador de bordo de um trem de carga no mundo real.

REFERÊNCIAS

ALVARES, Luis Otavio; SICHMAN, Jaime Simão. Introdução aos sistemas multiagentes. In: **XVII Congresso da SBC - Anais JAI'97**. 1997.

BELLIFEMINE, Fabio; CAIRE, Giovanni; GREENWOOD, Dominic. **Developing multi-agent systems with JADE**. John Wiley & Sons, 2007.

BERNARDO, Wesley. **Implementação do algoritmo inspirado no comportamento da colônia de formigas para a problemática da condução de trens de carga.** 105 f. Trabalho de Conclusão de Curso (Bacharelado em Ciência da Computação) - Universidade Tecnológica Federal do Paraná. Ponta Grossa, 2019.

BORGES, André Pinz. **Descoberta de regras de condução de trens de carga.** 2009. 123 f. Dissertação (Mestrado) - Curso de Ciência da Computação, Programa de Pós-Graduação em Informática, Pontifícia Universidade Católica do Paraná, Curitiba, 2009.

BORGES, André Pinz. **Uma contribuição para geração de políticas de ações para condução de trens de carga usando Raciocínio Baseado em Casos.** 2015. 220 f. Tese (Doutorado) - Curso de Ciência da Computação, Programa de Pós-Graduação em Informática, Pontifícia Universidade Católica do Paraná, Curitiba, 2015.

BRINA, Helvécio Lapertosa. **Estradas de ferro.** Minas Gerais: Livros Técnicos e Científicos, 1983. 260 p.

CASTRO, Lucas Fernando Souza de. **Modelagem e implementação de um sistema multiagente utilizando a plataforma JaCaMo para alocação de vagas em um estacionamento inteligente**. 2015. 78 f. Trabalho de Conclusão de Curso (Bacharelado em Ciência da Computação) - Universidade Tecnológica Federal do Paraná. Ponta Grossa, 2015.

COSTA, Augusto Cesar Pinto Loureiro da. **Expert-Coop:** Um ambiente para desenvolvimento de Sistemas Multi-Agentes Cognitivos. 1997. 108 f. Dissertação (Mestrado) - Curso de Engenharia Elétrica, Programa de Pós-Graduação em Engenharia Elétrica, Universidade Federal de Santa Catarina, Florianópolis, 1997.

CUSTODIO, Vinicius. **Implementação de Agente Inteligente, Modelado como um Veículo Autônomo, para Desvio de Obstáculos Estáticos.** 2017. 65 f. Trabalho de Conclusão de Curso (Bacharelado em Ciência da Computação) - Universidade Tecnológica Federal do Paraná. Ponta Grossa, 2017.

DORDAL, Osmar Betazzi et al. An Intelligent System for train overtaking using distributed coordination. **IECON 2013-39th Annual Conference of the IEEE Industrial Eletronics Society**, [s.l], p.2239-3334, nov.2013. IEEE.

DORDAL, Osmar Betazzi et al. Strong reduction in fuel consumption driving trains in bi-directional single line using crossing loops. I**EEE International Conference on Systems, Man, And Cybernetics**, [s.l], p.1597-1602, out.2011. IEEE.

FERBER, Jacques. **Multi-Agent Systems:** An Introduction to Distributed Artificial Intelligence. London: Addison-Wesley, 1999.

FIPA. **The Foundation for Intelligent Physical Agents.** Disponível em: <http://www.fipa.org/>. Acesso em: 10 ago. 2019

HEIJMEIJER, Alexis Van Haare. **Interligação entre a ferramenta de simulação SUMO e o projeto MAPS**. 2016. 80 f. Trabalho de Conclusão de Curso (Bacharelado em Ciência da Computação) - Universidade Tecnológica Federal do Paraná. Ponta Grossa, 2016.

HÜBNER, Jomi Fred. **Um modelo de reorganização de Sistemas Multiagentes**. 2003. 224 f. Tese (Doutorado) - Curso de Engenharia Elétrica, Departamento de Engenharia da Computação e Sistemas Digitais, Universidade de São Paulo, São Paulo, 2003.

JENNINGS, Nicholas; WOOLDRIDGE, Michael. **Agent technology:** Foundations, Applications, and Markets. [S.l]: Springer Science & Business Media, 1998.

JOHNSON, Steven. **Emergence:** The connected lives of ants, brains, cities, and software. New York: Scribner Book Company, 2001. 288 p.

O'HARE, Gregory; JENNINGS, Nicholas. **Foundations of Distributed Artificial Intelligence.** John Wiley & Sons Ltd., 1996. 592 p. (Sixth-Generation Computer Technology).

PEGÔ FILHO, Bolívar. **Logística e transportes no Brasil:** uma análise do programa de investimentos 2013-2017 em rodovias e ferrovias. Brasília: Livraria Ipea, 2016.

RICARTE, Ivan Luiz Marques. Programação Orientada a Objetos: uma abordagem com Java. **Universidade Estadual de Campinas**, Campinas, SP, Brasil, 2001.

SATO, Denise Maria Vecino et al. Lessons learned from a simulated environment for trains conduction. In: IEEE INTERNATIONAL CONFERENCE ON INDUSTRIAL TECHNOLOGY, 1., 2012, Atenas. **Proceedings… .** Atenas: 2012 IEEE INTERNATIONAL CONFERENCE ON INDUSTRIAL TECHNOLOGY, 2012. p. 533-538.

SILVA, Luiz Cláudio Nogueira da; MENDES NETO, Francisco Miltons; JÁCOME JÚNIOR, Luiz. MobiLE: Um ambiente multiagente de aprendizagem móvel para apoiar a recomendação sensível ao contexto de Objetos de Aprendizagem. In: SIMPÓSIO BRASILEIRO DE INFORMÁTICA NA EDUCAÇÃO, XXII., 2011, Aracaju. **Anais... .** Aracaju: SBIE, 2011. p. 254 - 263.

SILVA, Marcos Rafael da. **Um Agente Deliberativo Aplicado ao Apoio à Condução de Trens.** 2011. 103 f. Tese (Doutorado), Pontifícia Universidade Tecnológica do Paraná, Curitiba, 2011.

STEINER, Donald D. IMAGINE: an integrated environment for constructing distributed artificial intelligence systems. **Foundations of Distributed Artificial Intelligence**, v. 9, p. 345, 1996.

STREISKY, Matheus. **Aplicação de algoritmo para recuperação de casos no problema de condução de trens de carga.** 2019. 100f. Trabalho de Conclusão de Curso (Bacharelado em Ciência da Computação) - Universidade Tecnológica Federal do Paraná. Ponta Grossa, 2019.

SUGURI, Hiroki. A standardization effort for agent technologies: The foundation for intelligent physical agents and its activities. In: **Proceedings of the 32nd Annual Hawaii International Conference on Systems Sciences. 1999. HICSS-32. Abstracts and CD-ROM of Full Papers**. IEEE, 1999. p. 10.

SYCARA, Katia P. Multiagent Systems. **AI magazine**, v. 19, n. 2, p.79, 15 jun. 1998.

TEIXEIRA, Fábio V. **JADE**: Jade Agent Development Framework. Faculdade de Engenharia Elétrica e de Computação. v. 9, 2012.

TELECOM ITALIA. **JADE Site | Java Agent DEvelopment Framework**. Disponível em <https://jade.tilab.com>. Acesso em 05 mar. 2019.

WOOLDRIDGE, Michael. **An introduction to Multiagent Systems**. 2nd, e.d [S.1]: Wiley Publishing, 2009.

ANEXO A - Equações para a condução de trens de carga

EQUAÇÃO	FUNÇÃO	DESCRIÇÃO
$R_T = \sum_{i=0}^{n_l}(R_{nl} + R_{cl} + R_\gamma + R_i) + \sum_{i=0}^{n_v}(R_{nv} + R_{cv} + R_\gamma + R_i)$	$f_{Rt}(R_{nl}, R_{cl}, R_\gamma, R_i, R_{nv}, R_{cv})$ $f_{Rt}(T, ST, pos)$*	Resistência total do trem (em kgf.).
$R_{nl} = 1{,}3 + \frac{29}{w} + 0{,}03 \times v + \frac{0{,}0024 \times A \times v^2}{w \times n}$	$f_{R_n}(w, v, A, n)$	Resistência normal da locomotiva (em kgf.).
$R_{nv} = 1.3 + \frac{29}{w} + 0{,}045 \times v + \frac{0{,}0024 \times A \times v^2}{w \times n}$	$f_{R_v}(w, v, A, n)$	Resistência normal dos vagões (em kgf.).
$R_{cl} = 0{,}2 + \frac{100}{R} \times (br + b + 3{,}8)$	$f_{R_l}(R, br, b)$	Resistência de curva da locomotiva (em kgf.).
$R_{cv} = \frac{500 \times b}{R}$	$f_{R_{cv}}(b, R)$	Resistência de curva do veículo (em kgf.).
$R_i = 10 \times i$	$f_{R_i}(i)$	Resistência de rampa do veículo (em kgf.).
$R_\gamma = 4 \times \frac{v_F^2 - v^2}{\ell}$	$f_{R_\gamma}(v_F, v, \ell)$	Resistência inercial (em kgf.).
$F_t = \frac{273.24 \times 0.82 \times HP}{v}$	$f_{F_t}(HP, v)$	Esforço trator (em kgf.).
$F_{tm} = \frac{W \times f}{1 + (0{,}01 \times v)}$	$f_{F_{tm}}(W, f, v)$	Força tratora máxima (em kgf.).
$F_{ac} = F_t - R_T$	$f_{F_{ac}}(F_{tm}, R_T)$	Força de aceleração disponível (em kgf.).
$v_F = \sqrt{\left\lvert v^2 + \frac{(F_{ac} \times 20)}{(4 \times W)} \right\rvert}$	$f_{v_F}(v, F_{ac}, W)$	Velocidade final desejada para cada deslocamento de 20m (em km/h).
$\Delta\ell = 4 \times \frac{W \times (v_F^2 - v^2)}{F_{ac}}$	$f_{\Delta\ell}(W, v_F, v, F_{ac})$	Deslocamento (em metros).
$\Delta t = 7{,}2 \times \frac{\Delta\ell}{v_F + v}$	$f_{\Delta t}(\Delta\ell, v_F, v)$	Variação da duração da ação (em segundos).
$v_m = \frac{\Delta\ell}{\Delta t} \times 3{,}6$	$f_{v_m}(\Delta\ell, \Delta t)$	Velocidade média (em km/h).
$\Delta t_{final} = \sum_{k=1}^{n} \Delta t_k$	$f_{\Delta t_{final}}(\Delta t_k)$	Tempo gasto total da viagem (em segundos).
$\Delta\ell_{final} = \sum_{k=1}^{n} \Delta\ell_k$	$f_{\Delta\ell_{final}}(\Delta\ell_k)$	Deslocamento total da viagem (em metros).
$\gamma = F_{ac} \times \frac{g}{W}$	$f_\gamma(F_{ac}, g, W)$	Aceleração do trem.
$C = \frac{\Delta t}{60} \times cp$	$f_C(\Delta t, cp)$	Consumo nominal para um intervalo de tempo (em litros/minuto).

$LTKB = \frac{\sum C}{W \times \Delta\ell_{final}}$	$f_{LGTT}(\sum C, W, \Delta\ell_{\text{final}})$	Consumo de uma viagem (em litros por *tonelada* bruta *transportada* – LTKB).
$F_{fd} = pad \times v_F \times 1417{,}475$	$f_{F_{fd}}(v_F, pad), se\ v \leq 38{,}6km/h$	Força de frenagem dinâmica realizada pelos motores da locomotiva (em kgf.) em velocidade abaixo de 38.6km/h.
$F_{fd} = pad \times v_F^{-1.06} \times 921526{,}6$	$f_{F_{fd}}(v_F, pad), se\ v > 38{,}6km/h$	Força de frenagem dinâmica realizada pelos motores da locomotiva (em kgf.) em velocidade acima de 38.6km/h.
$F_{fa} = ar \times P_{cl} \times ra \times ef \times ca \times 10$	$f_{F_{fa}}(ar, P_{cl}, ra, ef, ca)$	Força de frenagem automática realizada pelos vagões de um trem (em kgf.).
$F_f = F_{fd} + F_{fa} - R_i$	$f_{F_f}(ar, P_{cl}, ra, ef, ca, v_F, pad, R_i)$	Força de frenagem total.

Legenda:

A: área do veículo (em *pés-quadrados*)

ar: área do êmbolo do cilindro de freio (em m^2)

b: bitola da linha (em metros)

br: base rígida do veículo

ca: coeficiente de atrito entre sapata de freio e roda

cp: consumo de combustível em determinado ponto de aceleração (em litros)

ef: eficiência da timoneira de freio (em %)

f: coeficiente de aderência entre os trilhos e rodas

g: aceleração da gravidade (em m/s²)

HP: potência do veículo (em cavalo de força)

i: porcentagem da inclinação da via a cada 100m

kgf: quilograma força

ℓ: deslocamento desejado (em metros)

n: número de eixos

n_l: número de locomotivas

n_v: número de vagões

pad: ponto de frenagem dinâmica

P_{cl}: pressão do cilindro de freio (em kPa)

pos: posição da cabeça do trem T sobre a ST.

R: raio da curva (em metros)

ra: relação da alavanca da timoneira de freio

ST: descrição de um trecho de via férrea onde o trem T está posicionado.

T: descrição de um trem em termos de locomotivas e seus vagões.

v: velocidade (em km/h)

v_F: velocidade desejada (em km/h)

w: peso do veículo (em toneladas)

W: peso do trem (em toneladas)